Course Transcript

Java SE 8 Programming: Interfaces, Lambda Expressions, Collections, and Generics

Interfaces and Lambda Expressions

Collections and Generics

Collections, Streams, and Filters

Practice: Java SE8 Programming Basics

Course Introduction

Learning Objective

After completing this topic, you should be able to

- *start the course*

1. Introduction to the course

Java is a programming language that allows programmers to create objects that can interact with other objects to solve problems. I'm Jason Row and in this course, I'll review interfaces and lambda expressions in Java. In addition, I'll explore generic classes and collections such as implementing ArrayList, Sets, HashMaps, and Stacks.

Interfaces

Learning Objective

After completing this topic, you should be able to

- *describe how to work with interfaces in Java*

1. Working with interfaces in Java

In this video, we're going to look at interfaces in Java. An interface is another way to define an abstract class; only, they can contain public abstract methods. By having these public abstract methods, you're setting the contract that any classes that implement this interface must adhere to. And they either fulfill the entire contract or they have to be declared as abstract themselves. And you'll see that your interface will list the methods, but they don't have any implementation code. The curly braces simply aren't there, and that's up to the classes that implement the interface to handle. And while the interfaces normally contain public methods, they can also have constant fields. Let's take a look at a little problem that has a company that sells three different types of products. Now they're all different products, but they need to access the same type of data in a similar manner. So we have three types of products. We have Crushed Rock, Red Paint, and Widgets. So each of them is measured differently, but they all still have to be able to calculate the sales price, the cost, and the profit.

[Heading: Interfaces. Java interfaces can be used as a reference type and is an essential component of many design patterns.]

So how can interfaces help solve this problem? Now this would be an appropriate way to create a `CrushedRock` class if we're only dealing with `CrushedRock`. However, notice that this class has a `weight` field and our other two classes would actually differ from this. So if we had a `RedPaint` class, it would have a `gallons` field. And if we had a `Widget` class, it would have a `quantity` field. So calculating the required data for the three classes is similar, but still different from each of the other ones. Now since we have multiple products, we can actually make use of a `SalesCalcs` interface. Then you have it specify what methods have to be implemented by our classes. The method signature actually specifies what's going to be passed in and what is returned. Now there are a few rules for...when you're creating interfaces. For your access modifiers, all the methods in the interface are going to be `public`. So even if you forget to declare them as

`public`, you can't declare methods as `private` or `protected` in an interface. So they have to be `public`. Because all methods are implicitly abstract, it's actually redundant. But you can still place the abstract modifier when you're declaring a method as being abstract. And because all interface methods are abstract, you can't provide any method implementation. So you can't even provide the empty set of braces that's handled by the class that implements the interface.

[Heading: Interfaces. A sample code to explain the CrushedRock class before interfaces is as follows: public class CrushedRock { private String name; private double salesPrice = 0; private double cost = 0; private double weight = 0; // In pounds public CrushedRock(double salesPrice, double cost, double weight) { this.salesPrice = salesPrice; this.cost = cost; this.weight = weight; } } A sample code to explain the SalesCalcs interface is as follows: public interface SalesCalcs { public String getName(); public double calcSalesPrice(); public double calcCost(); public double calcProfit(); }]

Now a class can only extend a single abstract class. However, a class can implement more than one interface in a comma-separated list at the end of the class declaration. So this is a way to get around that limitation of being able to extend a single abstract class. You create interfaces and your class can implement multiple interfaces. So here's our updated `CrushedRock` class and it `implements` our `SalesCalcs` interface. So on the `class` declaration line, the `implements` keyword specifies the `SalesCalcs` interface for this class. And each of the methods specified by our `SalesCalcs` interface have to be implemented. However, how the methods are implemented, it's going to differ from class to class. And the older requirement is that the method signature has to match. And this allows the cost or the sales price calculations to differ between classes, but you still have the same method signature. Now since `CrushedRock implements SalesCalcs`, and a `SalesCalcs` reference could be used to access the data of a `CrushedRock` object. So in our code, we've created two instances of our `CrushedRock` class. But the first one is a reference to `CrushedRock` objects, while the second is a `SalesCalcs` object. And we can actually call the `calcSalesPrice()` method on either one. And they use the same implementation of the method found in the `CrushedRock` class. So if all three of our classes implement the same interface, we can actually have a list of objects that are made up of our three classes. And we're still able to process them in the same way since each has to implement the same methods, just in their own specific fashion.

[Heading: Interfaces. A sample code to explain the updated CrushedRock class that implements SalesCalcs is as follows: public class crushedRock implements SalesCalcs{ private String name = "Crushed Rock"; ... // a number of lines not

*shown @Override public double calcCost(){ return this.cost * this.weight; } @Override public double calcProfit(){ return this.calcSalesPrice() - this.calcCost(); } } Any class that implements an interface can be referenced using that interface. A sample code that explains how the calcSalesPrice method can be referenced by the CrushedRock class or the SalesCalcs interface is as follows: CrushedRock rock1 = new CrushedRock(12, 10, 50); SalesCalcs rock2 = new CrushedRock(12, 10, 50); System.out.println("Sales Price: " + rock1.calcSalesPrice()); System.out.println("Sales Price: " + rock2.calcSalesPrice()); The output of the sample code is as follows: Sales Price: 600.0 Sales Price: 600.0 A sample code to explain that any class implementing an interface can be referenced using that interface is as follows: SalesCalcs[] itemList = new SalesCalcs[5]; ItemReport report = new ItemReport(); itemList[0] = new CrushedRock(12.0, 10.0, 50.0); itemList[1] = new CrushedRock(8.0, 6.0, 10.0); itemList[2] = new RedPaint(10.0, 8.0, 25.0); itemList[3] = new Widget(6.0, 5.0, 10); itemList[4] = new Widget(14.0, 12.0, 20); System.out.println("==Sales Report=="); for(SalesCalcs item:itemList){ report.printItemData(item); }]*

You can see our code in the slide here has an `itemList` with a number of `CrushedRock` objects. We have a `RedPaint` object and `Widget`, but they all still implement `SalesCalcs`. So in our `for` statement down at the bottom, we can go through each of the items in our `itemList` and have them referenced as `SalesCalcs` items or interfaces. And we pass that into our `printItemData`. So we're still able to make use of them as sales classes even if they were still declared or referenced as their own specific class file. Instead of having to write a method that prints the data from each class or does a check to see if it's this class type, then do this, or if it's this other class type, do something different. Our interface reference actually allows you to retrieve the data from all three of our classes that we've used in our example. So to summarize, interfaces are similar to abstract classes in that they outline the `public` methods that have to be coded by any class that implements the interface. And your classes can implement more than one interface at a time. And by referencing the interface, your code can call the methods of any object that has been implementing that interface.

[Heading: Interfaces. A sample code to explain that the utility class that references the interface can process any implementing class is as follows: public class ItemReport { public void printItemData(SalesCalcs item){ System.out.println("--" + item.getName() + " Report--"); System.out.println("Sales Price: " + item.calcSalesPrice()); System.out.println("Cost: " + item.calcCost()); System.out.println("Profit: " + item.calcProfit()); } }]

Adding Methods to an Interface

Learning Objective

After completing this topic, you should be able to

- *use default and static methods in Java interfaces*

1. Using default and static methods

Let's talk about adding methods to our interfaces. Now interfaces generally have abstract methods that have to be implemented by the classes that implement the interface. But in Java 8, it now adds default methods as a new feature. And then using the `default` keyword allows you to provide fully implemented methods to all implementing classes. So the above example on our slide shows how we have an `Item Report`, which could've been implemented as a separate class. It can now be fully implemented as a default method. And if we had three classes that implemented our `interface SalesCalcs` interface, they would automatically get a fully implemented, `getItemReport` method. Now the feature was added to simplify the development of APIs that rely heavily on interfaces. And before, simply adding a new method would break all implementing and extended classes. But now default methods can be added or changed without harming any of the API hierarchies. Before having a `default` method in our interface, the `for` loop in our code would have had an `Item Report` class with a `printItemData` method.

[Heading: Adding Methods to an Interface. A code sample in which the Item Report can be fully implemented as a default method is as follows: public interface SalesCalcs { ... // A number of lines omitted public default void printItemReport() { System.out.println("--" + this.getName() + " Report--"); System.out.println("Sales Price: " + this.calcSalesPrice()); System.out.println("Cost: " + this.calcCost()); System.out.println("Profit: " + this.calcProfit()); } } A code sample for an updated version of the item report using default methods is as follows: SalesCalcs[] itemList = new SalesCalcs[5]; itemList[0] = new CrushedRock(12, 10, 50); itemList[1] = new CrushedRock(8, 6, 10); itemList[2] = new RedPaint(10, 8, 25); itemList[3] = new Widget(6, 5, 10); itemList[4] = new Widget(14, 12, 20); System.out.println("==Sales Report=="); for(SalesCalcs item:itemList){ item.printItemReport(); }]

But now printing the report involves simply calling our `printItemReport` method that is in our `SalesCalcs` interface. Another feature that's new to Java 8

is the ability to have `static` methods in our interfaces. And this is purely a convenience feature, so now you can include a helper method like the one on our screen. And you place it in the interface instead of having it as a separate class. So now that we have our `printItemArray` `static` `void` method, we can simply call it by calling `SalesCalcs.printItemArray` and pass in our item list. So we no longer have need of another class. We can place that simply within our interface. And while it's not new in Java 8, constant fields are permitted within the interfaces. So when you declare a field and an interface, it will be implicitly declared as `public` `static` `final`. But you may be able to or you may want to put in those modifiers yourself, however they are redundant. Now to quickly summarize about adding methods to our interfaces, with Java 8, you have two new features that have been added to interfaces so that you can code default methods as well as static methods. And you could still declare constant fields in your interfaces.

[Heading: Adding Methods to an Interface. A code sample to create helper methods is as follows: public interface SalesCalcs { ... // A number of lines omitted public static void printItemArray(SalesCalcs[] items){ System.out. println(reportTitle); for(SalesCalcs item:items){ System.out.println("--" + item.getName() + " Report--"); System.out.println("Sales Price: " + item.calcSalesPrice()); System.out.println ("Cost: " + item. calcCost()); System.out.println("Profit: " + item.calcProfit()); } } A code sample to explain constant fields in interfaces is as follows: public interface SalesCalcs { public static final String reportTitle="\n==Static List Report=="; ... // A number of lines omitted]

Extending an Interface

Learning Objective

After completing this topic, you should be able to

- *describe how to extend an interface in Java*

1. Extending an interface in Java

In this video, we'll look at how you can extend an interface. So just as one class can extend another class, the same can be done with interfaces. You can have one interface that extends another and as you see in our example, we have a `WidgetSalesCalcs` interface and it extends our `SalesCalcs` interface. So if a class implements `WidgetSalesCalcs`, it then has to add code for the methods in our new `WidgetSalesCalcs` interface as well as the methods that are listed in our `SalesCalcs` interface. Now you can also have a class that both extends another class as well as implements an interface. So on our slide, we have our `class WidgetPro` and it extends our `Widget` class, but it also implements a `WidgetSalesCalcs` interface. And when you encounter this kind of situation, you have to remember that you first need to write code that extends the parent class first before you start writing code to implement the interface. So extending the parent class takes priority. Now in this video, we saw two situations where we were using the `extends` keyword. And the first was how an interface can extend another interface. And then we talked about how a class that extends a class and implements an interface. It actually requires you to first ensure that the extended class methods are coded before looking at implementing the methods from the interface.

[Heading: Extending an Interface. A code sample to extend an interface is as follows: public interface WidgetSalesCalcs extends SalesCalcs{ public String getWidgetType(); } A code sample in which classes can extend a parent class and implement an interface is as follows: public class WidgetPro extends Widget implements WidgetSalesCalcs{ private String type; public WidgetPro(double salesPrice, double cost, long quantity, String type) { super(salesPrice, cost, quantity); this.type = type; } public String getWidgetType() { return type; } }]

Using Java Interfaces

Learning Objective

After completing this topic, you should be able to

- *use Java interfaces in a Java application*

1. Adding Java interfaces

In this demo, we'll update an existing application by changing a few of the classes so that they implement an interface instead of just inheriting methods from a shared class. So we're going to take some of the methods from our `Account` class and we'll copy the method signatures to our `AccountOperations` interface. So we'll open up Account.java. And the methods we want are `getBalance()`, `deposit`, `withdraw`, and `getDescription()`. So we'll **Copy** those over to AccountOperations.java. Now we can remove the `abstract` keyword from these method signatures. The `toString()` method, we'll remove entirely. And we also have to remove the actual implementation code since we just want the method signatures. There we go, we have our four methods in our interface. Now we can update our `CheckingAccount` class in order to implement `AccountOperations` interfaces. We'll go to our `CheckingAccount` class. You see that it currently `extends` the `Account` class, but now we want to `implements` `AccountOperations`. We also want to do that with our `SavingsAccount` class.

[The NetBeans IDE 8.0.2 window is displayed. The menu bar includes menus such as File, Edit, View, Navigate, Source, Refactor, Run, and Debug. The left pane of the window includes two sections. The first section includes the following tabs: Projects, Files, and Services. The Projects tab is open. The second section includes the AccountOperationsNavigator tab. The Projects tab includes the Interfacebanking06-02Prac node, which further includes subnodes such as Source Packages, Test Packages, Libraries, and Test Libraries. The Source Packages subnode includes the com.example subnode. The com.example subnode includes subnodes such as Account.java, AccountOperations.java, Bank.java, BankOperations.java, CheckingAccount.java, Customer.java, CustomerReport.java, and SavingsAccount.java. The presenter clicks the Account.java node. As a result, the right pane includes the Account.java tabbed page and it includes the following tabs: Source and History. The Source tab is selected and includes the following code: 1 package com.example; 2 3 public abstract class Account { 4 5 protected

double balance; 6 7 public Account(double balance) { 8 this.balance = balance; 9 } 10 11 public double getBalance() { 12 return balance; 13 } 14 15 public void deposit(double amount) { 16 balance += amount; 17 } 18 19 @Override 20 public String toString() { 21 return getDescription() + ": current balance is " + balance; 22 } 23 24 public abstract boolean withdraw(double amount); 25 26 public abstract String getDescription(); 27 28 } The right pane also includes the Output - ImmutableClassDemo (run) tabbed page at the bottom. The presenter clicks the AccountOperations.java subnode in the left pane and the right pane includes the AccountOperations.java tabbed page along with the Account.java tabbed page. It includes the following code: 1 package com.example; 2 3 public interface AccountOperations { 4 5 } 6 He then enters the following code on the AccountOperations.java tabbed page: 4 public double getBalance() { 5 6 public void deposit(double amount); 7 8 9 public boolean withdraw(double amount); 10 11 public String getDescription(); 12 } 13 Next the presenter clicks the CheckingAccount.java node in the left pane and the right pane includes the CheckingAccount.java tabbed page. It includes the following code: 1 package com.example; 2 3 4 public class CheckingAccount extends Account { 5 6 private final double overDraftLimit; 7 8 public CheckingAccount(double balance) { 9 this(balance, 0); 10 } 11 12 public CheckingAccount(double balance, double overDraftLimit) { 13 super(balance); 14 this.overDraftLimit = overDraftLimit; 15 } He modifies the code 'public class CheckingAccount extends Account {' on line #4 as follows: public class CheckingAccount extends Account implements AccountOperations { He then selects the SavingsAccount.java subnode in the left pane. The SavingsAccount.java tabbed page is displayed in the right pane and includes the following code; 1 2 3 package com.example; 4 5 6 public class SavingsAccount extends Account{ 7 Double rateofinterest=0.06; 8 9 public SavingsAccount(double balance) { 10 super(balance); 11 }]

So again, it `extends Account`, we want to have it `implements AccountOperations`. Now we can go into our `Account` class file and actually remove the methods that are now marked in our interface. So we can get rid of `getDescription()` and `withdraw`. And we'll remove `getBalance()` and `deposit` as well. We need to change our `toString()` method. Since we no longer have a `getDescription()` method, we'll just remove that and change the string to `"Current balance is "` and the `balance`. Right, so now we can go to our `CheckingAccount` class and add the code for the methods that we have listed. So let's go to our `CheckingAccount` class again. So you'll notice that it currently has methods for `withdraw` and `getDescription()`. So we just need to add code for the remaining methods. It should also have an `@Override` marker. So `@Override`, so we'll `@Override getBalance()` and `deposit`.

[The Interfacebanking06-02Prac - NetBeans IDE 8.0.2 window is open. The SavingsAccount.java tabbed page is displayed in the right pane of the window. The presenter modifies the code 'public class SavingsAccount extends Account{' on line #6 as follows: public class SavingsAccount extends Account implements AccountOperations{ He clicks the Account.java subnode in the left pane and the Account.java tabbed page is displayed in the right pane. He then deletes the following codes: 24 public abstract boolean withdraw(double amount); 25 26 public abstract String getDescription(); Next he deletes the following lines of code: 10 11 public double getBalance() { 12 return balance; 13 } 14 15 public void deposit(double amount) { 16 balance += amount; 17 } He then modifies the code 'return getDescription() + ": current balance is " + balance;' that is now on line #14 as follows: 14 return "Current balance is " + balance; Next he clicks the CheckingAccount.java subnode in the left pane and the CheckingAccount.java tabbed page is displayed in the right pane. He scrolls down the page and it includes the following codes: 16 17 @Override 18 public boolean withdraw(double amount) { 19 if (amount <= balance + overDraftLimit) { 20 balance -= amount; 21 return true; 22 } else { 23 return false; 24 } 25 } 26 27 @Override 28 public String getDescription() { 29 return "Checking Account"; 30 } 31 32 } He then adds the following codes from line #32: 32 @Override 33 public double getBalance() { 34 return balance; 35 } 36 37 @Override 38 public void deposit(double amount) { 39 balance += amount; 40 } 41 42 43 @Override 44 public String toString() { 45 return this.getDescription() + " balance is " + balance; 46 }]

And we also have a `toString()` method that gets used as well. Then we open the `SavingsAccount` class file, and we need to implement the `getBalance()` and the `toString()` methods since the class already has methods for `deposit` and `getDescription()`. So I'll just **Copy** and **Paste** that code that we need. So there's `getBalance()` and we need to have `toString()` in here as well. Make sure you're choosing right quotation marks, and that works better. And we need another closing brace, there. Now we're going to do something similar with a `BankOperations` interface. And so we want to open up our Bank.java file. And we want to have it implement a `BankOperations` interface. So we have our `public class Bank` and we want to have it `implements BankOperations`.

[The Interfacebanking06-02Prac - NetBeans IDE 8.0.2 window is open. The CheckingAccount.java tabbed page is displayed in the right pane of the window. The presenter clicks the SavingsAccount.java subnode in the left pane and the SavingsAccount.java tabbed page is displayed in the right pane. He scrolls down the page and it includes the following code: 18 @Override 19 public boolean withdraw(double amount) { 20 if (amount <= balance) { 21 balance -= amount; 22 return true; 23 } else { 24 return false; 25 } 26 } 27 28 29 @Override 30 public void

*deposit(double amount) { 31 balance += amount; 32 balance +=balance * rateofinterest; 33 34 } 35 36 @Override 37 public String getDescription() { 38 return "Savings Account"; 39 } 40 41 } 42 He enters the following code from line #41: 41 @Override 42 public double getBalance() { 43 return balance; 44 } 45 46 @Override 47 public String toString() { 48 return this.getDescription() + " balance is " + balance; 49 50 51 } 52 } 53 Next the presenter clicks the Bank.java subnode from the left pane. The right pane includes the Bank.java tabbed page and displays the following code: 1 package com.example; 2 3 public class Bank { 4 5 private Customer[] customers; 6 private int numberOfCustomers; 7 8 public Bank() { 9 customers = new Customer[10]; 10 numberOfCustomers = 0; 11 } 12 13 public void addCustomer(String f, String l,Branch b) { 14 int i = numberOfCustomers++; 15 customers[i] = new Customer(f, l,b); He modifies the code 'public class Bank {' on line #3 as follows: public class Bank implements BankOperations {]*

Now we need to add our methods in first to our `BankOperations` interface. So I'll open up BankOperations.java, and I'll quickly **Paste** in the three methods that we need. And we need semicolons at the end of those. Now we need to copy a method to our interface and that's actually found in our `CustomerReport` class. And we have our `generateReport()` method, so let's **Copy** that. We'll go back to our `BankOperations` interface, we'll **Paste** that in. Now since we're actually putting code...an actual code in for our method in our interface, we need to have the `default` keyword. And that will make sure that it will use this method instead. And all these references to `bank`, change them to `this`, and this should be `AccountOperations`.

[The Bank.java tabbed page is open in the Interfacebanking06-02Prac - NetBeans IDE 8.0.2 window. The presenter selects the BankOperations.java node from the left pane and the right pane includes the BankOperations.java tabbed page. This page includes the following code: 1 package com.example; 2 3 public interface BankOperations { 4 5 } 6 The presenter enters the following code from line #5: 5 public void addCustomer(String f, String l,Branch b); 6 public int getNumOfCustomers(); 7 public Customer getCustomer(int customerIndex); 8 9 } He then selects the CustomerReport.java node from the left pane. The right pane includes the CustomerReport.java tabbed page. It includes the following code: 15 public void generateReport() { 16 17 // Print report header 18 System.out.println("\t\t\tCUSTOMERS REPORT"); 19 System.out.println("\t\t\t==================="); 20 21 // For each customer... 22 for (int custIndex = 0; custIndex < bank.getNumOfCustomers(); custIndex++) { 23 Customer customer = bank.getCustomer(custIndex); 24 25 // Print the customer's name 26 System.out.println(); 27 System.out.println("Customer: " 28 + customer.getLastName() + ", " 29 + customer.getFirstName() 30 + "\nBranch: " + customer.getBranch() + ", " 31 + customer.getBranch().getServiceLevel()); 32 33 //

For each account for this customer... 34 for (int acctIndex = 0; acctIndex< customer.getNumOfAccounts(); acctIndex++) { 35 Account account = customer.getAccount(acctIndex); 36 37 // Print the current balance of the account 38 System.out.println(" " + account); 39 } 40 } 41 } 42 } 43 The presenter copies the codes from line #15 to line #41. Next he clicks the BankOperations.java subnode from the left pane and the BankOperations.java tabbed page is displayed in the right pane. He pastes and modifies the code as follows: 9 public default void generateReport() { 10 11 // Print report header 12 System.out.println("\t\t\tCUSTOMERS REPORT"); 13 System.out.println("\t\t\t================"); 14 15 // For each customer... 16 for (int custIndex = 0; custIndex < this.getNumOfCustomers(); custIndex++) { 17 Customer customer = this.getCustomer(custIndex); 18 19 // Print the customer's name 20 System.out.println(); 21 System.out.println("Customer: " 22 + customer.getLastName() + ", " 23 + customer.getFirstName() 24 + "\nBranch: " + customer.getBranch() + ", " 25 + customer.getBranch().getServiceLevel()); 26 27 // For each account for this customer... 28 for (int acctIndex = 0; acctIndex< customer.getNumOfAccounts(); acctIndex++) { 29 AccountOperations account = customer.getAccount(acctIndex); 30 31 // Print the current balance of the account 32 System.out.println(" " + account); 33 } 34 } 35 }]

Looks good, and then we can actually **Delete** our CustomerReport.java file since it's no longer needed. So now we can run our application. And you can see that in the Output window, we have our "CUSTOMERS REPORT" that's been printed out. So for each "Customer" we get the "Branch" and how much their balance is. And everything is running successfully. So in this demo, we saw how we can create interfaces and how classes can extend one class, but also implement another interface as well. And we also saw the use of a default method within an interface.

[The BankOperations.java tabbed page is open in the Interfacebanking06-02Prac - NetBeans IDE 8.0.2 window. The presenter right-clicks the CustomerReport.java subnode from the left pane and selects the Delete option from the shortcut menu displayed. As a result, the Delete dialog box is displayed with the following checkboxes and the OK, Cancel, and Help buttons: Safely delete (with usage search) and Search In comments. He clicks the OK button to close the dialog box. The presenter then clicks the Run button on the toolbar of the Interfacebanking06-02Prac - NetBeans IDE 8.0.2 window and runs the program. As a result, the Output - Interfacebanking06-02Prac (run) tab displays the following partially visible output: Customer: Bryant, Owen Branch: Bangalore, Full Checking Account balance is 200.0 Customer: Soley, Tim Branch: LA, Basic Checking Account balance is 200.0 Customer: Soley, Maria Branch: Bangalore, Full Checking Account balance is 100.0 BUILD SUCCESSFUL (total time: 0 seconds)]

Anonymous Inner Classes

Learning Objective

After completing this topic, you should be able to

- *describe how to use anonymous inner classes in Java*

1. Using anonymous inner classes in Java

Let's now discuss anonymous inner classes. An anonymous inner class is simply a class that's been defined within your code instead of in a separate file. There are a few reasons for doing this. First it allows you to group your code in one place. If it's a short class file, you can keep it within an existing class instead of placing it in a separate .java file. It also increases encapsulation of your code. You can hide the class within another and prevent it from being used by other locations in your code. And using anonymous inner classes can actually make your code more readable by the next developer that has to review or fix it. So let's look at an interface called `StringAnalyzer`. And any class that implements this interface will need code for the `analyze` method and any interface with a single method can also be known as a "functional interface". So let's see how this could be a good choice for anonymous inner class. So let's say that we had a `ContainsAnalyzer` class that implemented our `StringAnalyzer` interface. We then have code that creates an instance of the `ContainsAnalyzer` class. We then say it would be used in the `searchArr` method. And within our `searchArr` method, there would have been a call to `contains.analyze`, in order to get the proper Boolean value. So now we'd have to jump between multiple classes to follow our code. And instead of the `ContainsAnalyzer` class, we could be using an anonymous inner class.

[Heading: Anonymous Inner Classes. The sample code to explain the StringAnalyzer interface is as follows: public interface StringAnalyzer { public boolean analyze (String target, String searchStr); } An anonymous inner class takes two strings and returns a boolean value. A sample code that explains the syntax of the method call with concrete class is as follows: 20 // Call concrete class that implements StringAnalyzer 21 ContainsAnalyzer contains = new ContainsAnalyzer(); 22 23 System.out.println("===Contains==="); 24 Z03Analyzer.searchArr(strList01, searchStr, contains);]

You'll notice that in our second set of our...or second batch of our code, we've the third parameter is now has `new StringAnalyzer`. And then within our curly braces, it has our overrides for the `analyze` method. That's our anonymous inner class. And it doesn't specify a name, but it implements basically the same code that would've been in our `ContainsAnalyzer` class. And the syntax might seem a little complicated as you are looking at this since you've a class being defined where a parameter variable would normally be. Now the code follows the reasons that I mentioned for anonymous inner classes. It's actually logically grouping code in one place since we only have a single method for our class. We're actually increasing encapsulation as the anonymous inner class is usable only from within the specific class. And it's certainly more readable. We don't have to dive into another class or a second or third class to see what our calls to `StringAnalyzer.analyze` method would be doing. It's all contained within our initial or our first class file.

[Heading: Anonymous Inner Classes. A sample code of the anonymous inner class is as follows: 22 Z04Analyzer.searchArr(strList01, searchStr, 23 new StringAnalyzer() { 24 @Override 25 public boolean analyze(String target, String searchStr) { 26 return target.contains(searchStr); 27 } 28 });]

Comparing Classes, Interfaces, and Lambda Expressions

Learning Objective

After completing this topic, you should be able to

- *describe how classes, interfaces, anonymous inner classes, and lambda expressions can be used to provide similar code functionality in Java*

1. Comparing a solution

In this video, let's compare a solution as it's been done first with just classes, then using interfaces, and an `anonymous inner class`, then lastly with lambda expressions. So we have a class on the screen here and we have a method that takes a source string and searches our text that matches the text in the search string. And if there is a match, a true result is being returned and if no match is found then it returns false instead. Now additional methods could be written to compare lengths or determine if the string starts with the search string. But for this one, only one method is implemented for simplicity. So here's our code to test our class and our first class is pretty standard. We have a test array that's passed into a `foreach` loop where our method is used to print out matching words. So our output ends up being matched `"tomorrow"`, matched `"toto"`, matched `"to"`, and matched `"timbukto"`. So it's a fairly simple way of testing our basic class file. Now in our next example, we're going to switch it up and we're going to use an interface instead of just a plain class. So you see that we now have an `interface` called `StringAnalyzer` and note that `StringAnalyzer` is a functional interface. It only has that one method.

[Heading: Comparing Classes, Interfaces, and Lambda Expressions. A class analyzes an array of strings given a search string. It can be used to print strings that contain the search string. Other methods could be written to perform a similar string test. A sample code of the regular class method is as follows: 1 package com.example; 2 3 public class AnalyzerTool { 4 public boolean arrContains (String sourceStr, String searchStr) { 5 return sourceStr.contains (searchStr); 6 } 7 } 8 A sample code to test the Z01Analyzer class is as follows: 4 public static void main (String[] args) { 5 String[] strList = 6 {"tomorrow","toto","to","timbukto","the","hello","heat"}; 7 String searchStr = "to"; 8

System.out.println("Searching for: " + searchStr"); 9 10 // Create regular class 11 AnalyzerTool analyzeTool = newAnalyzerTool(); 12 13 System.out.println("===Contains==="); 14 for(String currentStr:strList) { 15 if (analyzeTool.arrContains (currentStr, searchStr)) { 16 System.out.println("Match: " + currentStr); 17 } 18 } 19 } A sample StringAnalyzer code is as follows: 3 public interface StringAnalyzer { 4 public boolean analyze(String sourceStr, String searchStr); 5 } The StringAnalyzer is a single method functional interface.]

Now other than the addition of an `implements` clause, really the class hasn't changed that much. And if we replace our previous example and implement the interface, we end up with `public class ContainsAnalyzer implements StringAnalyzer`. So it's still basically the same class, we have just started using it in an interface instead. Now if we look back to our test class, the change to our interface or the change of using an interface doesn't really change our test class much at all. And the only difference is that a different class is being used to do that string testing. And also if additional tests need to be performed, then this would require additional `foreach` loops and a separate class for each test condition. So this could be seen as a step back. However, there are a couple of advantages to this approach. Now by encapsulating a `for` loop into a static helper method, only one loop is needed to process any sort of string test using the `StringAnalyzer` interface. And the `searchArr` method actually remains unchanged in all the examples that follow. Now do note that the parameters for our methods are we have one, a string array; the second parameter is the `searchStr`; and for three, we have a class that implements the `StringAnalyzer` interface.

[Heading: Comparing Classes, Interfaces, and Lambda Expressions. A sample code that is used to implement an interface in a StringAnalyzer code is as follows: 3 public class ContainsAnalyzer implements StringAnalyzer { 4 @Override 5 public boolean analyze (String target, String searchStr) { 6 return target.contains (searchStr); 7 } 8 } A sample code of the StringAnalyzer interface test class is as follows: 4 public static void main (String[] args) { 5 String[] strList = 6 {"tomorrow","toto","to","timbukto","the","hello","heat"}; 7 String searchStr = "to"; 8 System.out.println ("Searching for: " + searchStr); 9 10 // Call concrete class that implements StringAnalyzer 11 ContainsAnalyzer contains = new ContainsAnalyzer(); 12 13 System.out.println("===Contains==="); 14 for(String current:strList) { 15 if (contains.analyze(currentStr, searchStr)) { 16 System.out.println("Match: " + currentStr); 17 } 18 } 19 } A sample code to encapsulate a 'for' loop is as follows: 3 public class Z03Analyzer { 4 5 public static void search searchArr(String[] strList, String searchStr, StringAnalyzer analyzer) { 6 for(String currentStr:strList) { 7 if (analyzer.analyze(currentStr, searchStr)) { 8 System.out.println("Match: " + currentStr); 9 } 10 } 11 } // A number of lines omitted]

So now with the help of our helper method, our `main` method actually shrinks down. And the array can be searched and the results displayed with a single call on line 23. So we have `Z03Analyzer.searchArr` and then in brackets, we have a `strList01`, then the `searchStr`, and our `contains` variable which is...`ContainsAnalyzer` class that implements our `StringAnalyzer`. Now in our next example, we've changed our third argument to the method call into an `anonymous inner class`. So if you notice, our class structure is the same as the `ContainsAnalyzer` class in previous examples. But using this approach, our code is stored right here in the calling class. And in addition the logic for the `analyze` method can easily be changed depending on the circumstances. However, there are a couple of drawbacks. Now the syntax does seem a little complicated at first. And our entire class definition is included within the parentheses of the argument list.

[Heading: Comparing Classes, Interfaces, and Lambda Expressions. A sample code for the StringAnalysis test class with the helper method is as follows: 13 public static void main (String[] args) { 14 String[] strList01 = 15 {"tomorrow","toto","to","timbukto","the","hello","heat"}; 16 String searchStr = "to"; 17 System.out.println ("Searching for: " + searchStr); 18 19 // Call concrete class that implements StringAnalyzer 20 ContainsAnalyzer contains = new ContainsAnalyzer(); 21 22 System.out.println ("===Contains==="); 23 Z03Analyzer.searchArr(strList01, searchStr, contains); 24 } The sample code for the StringAnalysis anonymous inner class is as follows: 19 // Implement anonymous inner class 20 System.out.println ("===Contains==="); 21 Z04Analyzer.searchArr(strList01, searchStr, 22 new StringAnalyzer() { 23 @Override 24 public boolean analyze(String target, String searchStr) { 25 return target.contains(searchStr); 26 } 27 }); 28 }]

Now since there's no class name, when the code is compiled, a class file is going to be generated and a number assigned for the class. And this won't be a problem if there's only one `anonymous inner class`, but when there's more than one in multiple class or classes, it's going to be difficult to figure out which class file goes with which source file. Now with Java 8, a lambda expression can be substituted for an `anonymous inner class`. And let's take a look at some key facts. So the lambda expression has two arguments. So just like in our two previous examples, our lambda expression uses the same arguments as our `analyze` method. And the lambda expression returns a Boolean, so just like our two previous examples, our Boolean is returned just like our `analyze` method. And in fact the lambda expression, `anonymous inner class`, and the `concrete` class are all essentially equivalent. A lambda expression is simply a new way to express code logic using a functional interface as a base.

[Heading: Comparing Classes, Interfaces, and Lambda Expressions. The sample code for the StringAnalysis lambda expression is as follows: 13 public static void main (String[] args) { 14 String[] strList = 15 {"tomorrow","toto","to","timbukto","the","hello","heat"}; 16 String searchStr = "to"; 17 System.out.println ("Searching for: " + searchStr); 18 19 // Lambda Expression replaces anonymous inner class 20 System.out.println ("===Contains==="); 21 Z05Analyzer.searchArr(strList, searchStr, 22 (String target, String search) -> target.contains(search)); 23 }]

Lambda Expressions

Learning Objective

After completing this topic, you should be able to

- *describe how to use lambda expressions in Java*

1. Using lambda expressions in Java

In this video, we'll take a closer look at how to write lambda expressions. Now first off, a lambda expression is an argument list followed by our arrow token and then you have a block or expression. And when a code block is used, you can have multiple statements included in that block. And you can specify the parameter types as well as is done in our example where we specify that `x` and `y` are integers or how we have asked to specify it as a `String`. But you can also permit Java to infer the parameter type, which is why we have other examples where we don't show the parameter type for `x` and `y`, nor for `s`. We just have s as the arrow token, `s.contains("word")`. Java will infer the parameter types. So let's take a look at comparing a lambda expression to an implementation of a `StringAnalyzer` interface. So on our slide, we've a lambda expression and we're comparing it to an implementation of the `StringAnalyzer` interface. So in essence, they are equivalent because the lambda expression could be substituted for an anonymous inner class or the implementing class that we have here. And all three cases rely on our `StringAnalyzer` interface as our underlying plumbing. But you'll notice that we have our `analyze` method and it has two parameters, which our lambda expression also has the same or similar two parameters.

[Heading: Lambda Expressions. A table providing lambda expression definition includes a single row and the following three columns: Argument List, Arrow Token, and Body. The Argument List can be (int x, int y), the Arrow Token is ->, and the Body can be x + y. The basic lambda examples are as follows: (int x, int y) -> x + y (x, y) -> x + y (x, y) -> { system.out.println (x+y);} (String s) -> s.contains ("word") s -> s.contains ("word") The syntax for a lambda expression is as follows: (t,s) -> t.contains(s) The sample code of the ContainsAnalyzer.java class is as follows: public class ContainsAnalyzer implements StringAnalyzer { public boolean analyze (String target, String searchStr) { return target.contains(searchStr); } } Comparing the 'public boolean analyze (String target, String searchStr) { return target.contains(searchStr);' code statement with the lambda expression '(t,s) ->

t.contains(s)' the following inferences can be made: (String target, String searchStr) is equivalent to (t,s) return target.contains(searchStr); is equivalent to t.contains(s)]

And besides both our implementing class and our lambda expression having the same similar parameters, both have a body with one or more statements. In our lambda expression, we've `t.contains`, our `(s)` variable. We also have in our `analyze` method, we've `return target.contains(searchStr);` variable. Now our first example here is equivalent to the previous code example. The only difference is the lambda expression shorthand that is being used. And the type of arguments is inferred from the context of our lambda expression that it is used in. So our compiler knows that the signature for our `StringAnalyzer.analyze` would be `public Boolean analyze(String sourceStr, String searchStr)`. Thus the two strings are passed in and a Boolean is returned. Now if you notice our second example on line 28, now it really becomes trivial to change the logic for a functional interface.

[Heading: Lambda Expressions. A sample code that uses shortened syntax in lambda expressions is as follows: 20 // Use short form Lambda 21 System.out.println("===Contains===") 22 Z06Analyzer.searchArr (strList01, searchStr, 23 (t,s) -> t.contains (s) 24 25 // Changing logic becomes easy 26 System.out.println("==Starts With=="); 27 Z06Analyzer.searchArr (strList01, searchStr, 28 (t,s) -> t.startsWith(s)); The code sample including the searchArr method arguments for the above code is as follows: public static void searchArr (String[] strList, String searchStr, StringAnalyzer analyzer)]

So we've instead of `t.contains(s)`, we can have `t.startsWith(s)`. And finally, lambda expressions can be treated like variables. They could be assigned, they could be passed around, and they can be reused. So we've two lambda expressions that are assigned to our `StringAnalyzer`. So `StringAnalyzer contains` and `StringAnalyzer startsWith`, so those values get assigned to or the value can get assigned to our variables. And then we can make use of them in...later on in our calls to the `searchArr` functions. Now we've seen how to use lambda expressions as variables. We also looked at how you create basic lambda expressions in Java.

[Heading: Lambda Expressions. The sample code where lambda expressions are treated as variables is as follows: 19 // Lambda expressions can be treated like variables 20 StringAnalyzer contains = (t,s) -> t.contains(s); 21 StringAnalyzer startsWith = (t,s) -> t.startsWith(s); 22 23 System.out.println("===Contains===") 24 Z07Analyzer.searchArr(strList01, searchStr, 25 contains); 26 27 System.out.println("===Starts With==="); 28 Z07Analyzer.searchArr (strList01, searchStr, 29 startsWith);]

Writing Lambda Expressions

Learning Objective

After completing this topic, you should be able to

- *write lambda expressions for a Java application*

1. Adding lambda expressions in Java

Let's use this demo to see how we can add a few additional lambda expressions to our string analyzer project. So first, we have our LambdaTest.java file open. And our `main` method already has a `String[]` array that's been declared with a number of words. And also, it creates an instance of our `AnalyzerTool`. And then we have our `searchStr`, which is the word `"to"`. Now it already checked to see if any of the words in our array contain our search string. It also checks to see if it `("==Starts With==")` our search string or even equals our search string. So now let's look at checking an array in the lambda expression to check to see if any of our words end with our search string. So we're going to have our `stringTool.showResult` method, it's what's going to be called. Into our `showResult` method, we pass in our `stringList01`, our `searchStr`, and we also pass in our lambda array. In this case, I'll add in our argument list. So we're going to use `t` and `s`. We then have our arrow, `->`, token. And now it's time for our code block. And we can make use of the `endsWith` method call of our string.

[The LambdaBasics06-03Prac - NetBeans IDE 8.0.2 window is displayed. The menu bar includes the menus such as File, Edit, View, Navigate, Source, and Help. The toolbar is placed below the menu bar and includes icons such as Open, New, Undo, Redo, and Start. The window includes two panes. The left pane of the window includes two sections placed one below the other. The upper section includes the following tabs: Projects, Files, and Services. The Projects tab is currently open. The lower section includes the main - Navigator tab. The Projects tab includes the LambdaBasics06-03Prac project, which is expanded further and includes the following nodes: Source Packages, Test Packages, Libraries, and Test Libraries. The Source Packages node includes the com.example package. The com.example package includes the following class files: AnalyzerTool.java, LambdaTest.java, Main.java, and StringAnalyzer.java. The main - Navigator tabbed page in the lower section of the left pane includes the LambdaTest node with the main(String[] args) subnode. The right pane includes two sections. The upper section includes the LambdaTest.java tabbed page, which contains the following

tabs: Source and History. The Source tab is selected by default and displays the following code: 1 package com.example; 2 3 public class LambdaTest { 4 public static void main(String[] args) { 5 String[] strList01 = 6 {"tomorrow","toto","to","timbuktoo","the","hello","heat"}; 7 8 AnalyzerTool stringTool = new AnalyzerTool(); 9 String searchStr = "to" 10 11 System.out.println("Searching for: " + searchStr); 12 13 System.out.println("==Contains==") 14 stringTool.showResult(strList01, searchStr, 15 (t,s) -> t.contains (s) 16 17 System.out.println("==Starts With=="); 18 stringTool.showResult(strList01, searchStr, 19 (t,s) -> t.startsWith(s)); 20 21 System.out.println("==Equals=="); 22 stringTool.showResult(strList01, searchStr, 23 (t,s) -> t.equals(s)); 24 25 System.out.println("==Ends With=="); 26 // Your code here 27 28 System.out.println("==Less than 5=="); 29 // Your code here 30 31 System.out.println("==Greater than 5=="); The lower section of the right pane includes the Output - LambdaBasics06-03Prac (run) tabbed page. The presenter enters the following code from line #26 after the 'System.out.println("==Ends With==");' code statement on line #25: 26 stringTool.showResult(strList01, searchStr, 27 (t,s) -> t.endsWith(s));]

So we're going to call `t`, which would be our string list, that's what we iterate through in our lambda expression. So for each element in our array, it's going to call `t.endsWith` and it's going to check for our search string. Now we can do something similar for the...checking that our word has `("==Less than 5==")` characters. So once again, we have our `stringTool.showResult` method call. And in this case, our lambda expression is going to be `(t,s)`. And then we're going to call `t.contains(s)`, so we want to make sure that it does contain our search string. But we're also going to add another condition on here, that's going to call `t.length() < 5`. And now we want to check to see that if our word contains our search string, we want to make sure that it only does that for words that are `("==Greater than 5==")`. So we can do something pretty much the same again, our argument list `(t,s)`-

[The LambdaTest.java tabbed page is displayed in the LambdaBasics06-03Prac - NetBeans IDE 8.0.2 window. The presenter enters the following code from line #30 after the 'System.out.println("==Less than 5==");' code statement on line #29: 30 stringTool.showResult(strList01, searchStr, 31 (t,s) -> t.contains(s) && t.length() < 5); He then scrolls down the code on the LambdaTest.java tabbed page and enters the following code statement on line #34 after the 'System.out.println("==Greater than 5==");' code statement on line #33: stringTool.showResult(strList01, searchStr,]

`-> t.contains(s)`, so we'll make use of that `contains` method call again. And in this case, we want `t.length() > 5`. So pretty much exactly the same,

we're just increasing the `length()` condition on our lambda expression. So now if we run our application, we'll see in our Output window, we're searching for "to". We have when it "==Contains=="...the words that contain the search string to are "tomorrow", "toto", "to", and "timbuktu". Also it checks to see if it "==Starts With=="...so we have "tomorrow", "toto", and "to". There's only one that "==Equals==" in our string array, that's "to". Then we also have our "==Ends With==", and "==Less than 5==", and "==Greater than 5==". So our code does work. So in this video, we took a look at lambda expressions. So we saw how to write a lambda expression by adding an argument list, having an arrow token, and then adding our code block. And we made use of some of the methods available in our `String` class. So we called `endsWith`, `contains`, and the `length()` methods in order to perform our checks or in the creation of our lambda expressions.

[The LambdaTest.java tabbed page is displayed in the LambdaBasics06-03Prac - NetBeans IDE 8.0.2 window. The presenter enters the following code statement on line #35: (t,s) -> t.contains(s) && t.length() > 5); The presenter clicks the Start icon on the toolbar to run the program. As a result, the following output is displayed on the Output - LambdaBasics06-03Prac (run) tabbed page in the lower section of the right pane: Searching for: to ==Contains== Match: tomorrow Match: toto Match: to Match: timbukto ==Starts With== Match: tomorrow Match: toto Match: to ==Equals== Match: to ==Ends with== Match: toto Match: to Match: timbukto ==Less than 5== Match: toto Match: to ==Greater than 5== Match: tomorrow Match: timbukto BUILD SUCCESSFUL (total time: 1 second)]

Creating a Custom Generic Class

Learning Objective

After completing this topic, you should be able to

- *describe how to create a custom generic class using the type inference diamond*

1. Using the type inference diamond

In this video, we're going to discuss the use of generics in Java and how to use generic objects in your code. Now one of the most important features about generics in Java is that it provides type safety for your code. You can have types or methods that can operate on objects of various types while giving compile-time type safety. Now besides being easier to write, you also won't have to cast objects, especially when you're dealing with collections of objects. And you'll notice that the Java Collections API makes quite a number of uses of generics. So let's start with a couple of simple classes that don't make use of generics. So we have two examples in our slide and they're showing some simple caching objects. So even though each class is very simple, a separate class is required for each object type. So we have a `class CacheString` and a `class CacheShirt`, and they both have `private String` variables along with an `add` and a `get()` method. So to create a generic version of a `CacheAny` class, a variable named `<T>` is added to our `class` definition and it's surrounded by angle brackets. And in this case, `<T>` stands for type. And it can represent any type.

[Heading: Creating a Custom Generic Class. The code examples for simple cache class without generics are as follows: Code 1: public class CacheString { private String message; public void add(String message) { this.message = message; } public String get() { return this.message; } } Code 2: public class CacheShirt { private Shirt shirt; public void add(Shirt shirt) { this.shirt = shirt; } public Shirt get() { return this.shirt; } } The code sample for a generic cache class is as follows: 1 public class CacheAny <T>{ 2 3 private T t; 4 5 public void add (T t) { 6 this.t = t; 7 } 8 9 public T get() { 10 return this.t; 11 } 12 }]

As an example in our slideshow shows, the code is changed to use `T` instead of a specific type of information. And this change allows the `CacheAny` class to store any type of object. And `T` was chosen not by accident, but actually by convention. And a number of letters are commonly used with generics. Now you can use any

identifier you want, but the following values are merely strongly suggested. So you can also use `T` for type, which we were doing in this case, `E` for element, `K` for key, `V` for value. And we also use `S` and `U`. And they're used if there are second types, or third types, or more. Now here we have our generics in action. So you can note how the one generic version of our class can actually replace any number of type-specific caching classes. We no longer have a `CacheShirt` or a `CacheMessage` class. We're now using...for generics, we have `CacheAny`, then we state the name or the type of our object that's going to be stored in our `CacheAny` class. In the first case, we're going to have a string, which `myGenericMessage` is. `myGenericShirt` is a shirt type.

[Heading: Creating a Custom Generic Class. The code sample for generics in action is as follows: 1 public static void main(String args[]) { 2 CacheString myMessage = new CacheString(); // Type 3 CacheString myShirt = new CacheShirt(); // Type 4 5 //Generics 6 CacheAny<String> myGenericMessage = new CacheAny<String>(); 7 CacheAny<Shirt> myGenericShirt = new CacheAny<Shirt>(); 8 9 myMessage.add("Save this for me"); //Type 10 myGenericMessage.add("Save this for me"); // Generic 11 12 }]

Now our `add` and `get()` functions will work exactly the same way. And in fact, if my...if the `myMessage` declaration is changed to generic, no changes need to be made to the remaining code. Now the type inference diamond is or was a new feature in JDK 7. And in the generic code, we can look and see how the right side of our type definition is always equivalent to the left-side type definition. But in JDK 7, you can use a diamond to indicate that a right type definition is equivalent to the left. So on our screen, we have our `CacheAny`. And then in our angle brackets, we have `<String> myMessage = new CacheAny<>();` and then we don't repeat the `<String>` type, we merely use our inference diamond. And Java will know that on the right side, it should equal the type used on the left side. And this is really to help avoid typing redundant information over and over again. If you have longer type names or longer class names, it's just adding to how much typing you have to do. But...and in a way, this is working in the opposite way from a normal Java-type assignment. For example, if you have an `Employee emp = new Manager`, that's going to make your `emp` object an instance of `Manager`, so your right side dictates what the left side is going to be. But in the case of generics, if we have our left side of the expression rather than the right side, it's going to determine the type. Right, so in this video, we've seen some of the general reasons for using generics and the basic way of creating a class that makes use of generics.

[Heading: Creating a Custom Generic Class. The code example for generics with type Inference diamond is as follows: //Generics CacheAny<String> myMessage = new CacheAny<>();]

Overview of Collections

Learning Objective

After completing this topic, you should be able to

- *describe the Java Collections Framework*

1. Describing Java collections

Now a collection is really a single object that manages a group of other objects, and objects in the collection are called elements. And various collection types implement standard data structures, including stacks, queues, dynamic arrays, and even hashes. And all the collection objects have been optimized for use in Java applications. Now the collection classes are all stored in the `java.util` package. And we don't show all `import` statements in the following examples, but the `import` statements required for each collection type would be `import java.util.List`, `import java.util.ArrayList`, or `import java.util.Map`. Those are the most common ones. So our diagram on the screen shows the collection framework and the framework is made up of a set of interfaces for working with our groups or our collection of objects. Now we've `List`, `Set`, and `Map` and those are the interfaces of Java. And with all the concrete implementations of them are available within the `Collection` API. So if you look at the top we've our `Collection` interface and underneath that we've our `Set` and `List` interface. And after the site we've `Map`, which is a little different but or a little different implementation. But it's still part of our normal collection types.

[Heading: Overview of Collections. Primitive data types are not allowed in Collections. The Collections API relies heavily on generics for its implementation. A diagram which displays the collection types framework is displayed. It includes the following interfaces: <Interface> Collection, <Interface> Set, <Interface> List, <Interface> SoretedSet, and <Interface> Map. The <Interface> SoretedSet is connected to <Interface> Set. The <Interface> Set is connected to <Interface> Collection. The <Interface> List is connected to <Interface> Collection. The <Interface> Map is a separate element and not connected to any other element. The collection type Hashset is connected to the <Interface> Set. The collection type Treeset is connected to the <Interface> SoretedSet. The Arraylist and LinkedList collection types are connected to <Interface> List.]

Now from those `Set` and `List` interfaces, we can then start looking at the actual code that implements our interfaces. So here is a table that shows our `Collection` Interfaces and the actual Implementation code. So we talked about how we have `List`, `Set`, `Map` for interfaces. There's also a new one called `Deque` that's one of the new interfaces. And the code that are the classes that implement these interfaces. For List, we've `ArrayList` and `LinkedList` classes. For the `Set` interface, we've our `TreeSet`, `HashSet`, and `LinkedHashSet` classes. Then for `Map`, we've `HashMap`, `HashTable`, and `TreeMaps` and then for our `Deque` interface, we've `ArrayDeque` as the class that does the actual implementation code.

[Heading: Overview of Collections. A table that includes information about Collection Interfaces and Implementation is displayed. The table includes four rows and the following two columns: Interface and Implementation. For the List Interface, the Implementation values are ArrayList and LinkedList classes. For the Set Interface, the Implementation values are TreeSet, HashSet, and LinkedHashSet classes. For the Map Interface, the Implementation values are HashMap, HashTable, and TreeMaps classes. For the Deque Interface, the Implementation value is ArrayDeque class.]

Implementing an ArrayList

Learning Objective

After completing this topic, you should be able to

- *contrast the implementation of an ArrayList in Java without and with generics*

1. Implementing the list interface

In this video, we're going to cover the list interface and its role with collections. Now the list interface is the generic list behavior that we expect in lists or collections. And it's an ordered collection of elements. And it also includes things like adding elements at a specific index, getting an element based at an index, removing elements, overriding elements, and even getting the size of our list. Now one of the things about lists is that they do allow duplicate elements. Now our `ArrayList` is an implementation of our list interface. And one of the nice things about `ArrayList` is that it will automatically grow as you add elements and exceed the initial size that you thought your lists would be. It also has a numeric index, so it allows you to access the elements based on that index. So if you want to know the first, or the last, or the second, or the fifth element in the ArrayList, you can use those values or the index in order to get that information you're looking for.

You can also insert those elements based on the index, which also allows them to be overwritten. So we have some code here that has a `List` of integers, and our variable name is `partList`, and it's declared as a `new ArrayList` of type `<Integer>` since it's listed on the left-hand side, and the size of our `ArrayList` is going to be `3`. But then, we have four calls to the `add` method. So as we make...reach that fourth call to the `add` method, our list will automatically grow it in order to accept that you won't get an index that are ranging...an exception being thrown. Now we can call `partList.get` and pass in the index number of which part we want to get in this case. And it says `partList.get` in bracket `(0)`, so 0 is the first item in our array. Now we can also call the `.add` method once again and then pass in an index where we want our new element to be added in our `ArrayList`. Now let's briefly talk about autoboxing and unboxing and these are meant to make things simpler syntax-wise and have easier to recode for the next developer that comes along and looks at your code. Now if we look at lines 9 and 10, they, kind of, show a traditional method for moving between objects and primitives.

[Heading: Implementing an ArrayList. The sample code that explains the concept of array list integers is as follows: List<Integer> partList = new ArrayList<>(3); partList.add(new Integer(1111)); partList.add(new Integer(2222)); partList.add(new Integer(3333)); partList.add(new Integer(4444)); // ArrayList auto grows System.out.println("First Part: " + partList.get(0)); // First item partList.add(0, new Integer(5555)); // Insert an item by index The sample code that illustrates the concept of autoboxing and unboxing is as follows: 1 public class AutoBox { 2 public static void main(String[] args) { 3 Integer intObject = new Integer(1); 4 int intPrimitive = 2; 5 6 Integer tempInteger; 7 int tempPrimitive; 8 9 tempInteger = new Integer(intPrimitive); 10 tempPrimitive = intObject.intValue(); 11 12 tempInteger = intPrimitive; // Auto box 13 tempPrimitive = intObject; // Auto unbox]

But then we have in lines 12 and 13 in our code, which show boxing and unboxing. Now autoboxing and unboxing are Java language features that enable you to make sensible assignments without formal casting syntax. And Java provides a cast for you at compile time. But you need to be careful when you are using autoboxing in loops. There is actually a performance cost to using this feature. So in the example on our slide, we have a `partNumber` list that's created using an ArrayList. And there is no type definition when using syntax prior to Java version 1.5, so any type can be added to the list that's shown in line 8, where we have `partList.add("Oops a string!");`. Now it's up to the programmer to know what objects are in the list and in what order. So if our list was only for integer objects, a runtime error is going to occur at line 12. Now on lines 10-16, with a nongeneric collection, an `Iterator` is used to iterate through the list of items. And you'll notice that there is a lot of casting required to get the objects back out of the list, so you can print the data.

[Heading: Implementing an ArrayList. The following sample code illustrates the concept of array list without generics is as follows: 1 public class OldStyleArrayList { 2 public static void main(String args[]) { 3 List partList = new ArrayList(3); 4 5 partList.add(new Integer(1111)); 6 partList.add(new Integer(2222)); 7 partList.add(new Integer(3333)); 8 partList.add("Oops a string!"); 9 10 Iterator elements = partList.iterator(); 11 while (elements.hasNext()) { 12 Integer partNumberObject = (Integer) (elements. 13 int partNumber = partNumberObject.intValue(); 14 15 System.out.println("Part number: " + partNumber); 16 } 17 } 18 } The following code is a Java example that uses syntax prior to Java 1.5: 1 public class OldStyleArrayList { The following code is a Runtime error:ClassCastException: 13 int partNumber = partNumberObject.intValue();]

Now in the end, there's a lot of needless and extra code working with collections in this way. So if the line that adds our string to the `ArrayList` is commented out, our program would actually produce the output "Part number: 1111", "Part number: 2222", and "Part number: 3333". Now when we start using generics with our

ArrayList, things get a lot simpler. So when our ArrayList is initialized on line 3, any attempt to add an invalid value, such as we do at line 8 where we have partList.add("Bad Data");. And that's going to result in a compile time error now and on line 3, our ArrayList is assigned to a list type. And using this style enables you to swap out the list implementation without changing other code. So we always have a list, but we can always change if it's going to be ArrayList or not. Now we're using a foreach loop. It's actually much easier when we have...when we're iterating over items in our ArrayList. And it actually provides a lot cleaner code, so we don't have to actually cast anything to our elements because of the autounboxing features of Java.

[Heading: Implementing an ArrayList. The sample code that illustrates the concept of generic array list is as follows: 1 public class GenericArrayList { 2 public static void main(String args[]) { 3 List<Integer> partList = new ArrayList<>(3); 4 5 partList.add(new Integer(1111)); 6 partList.add(new Integer(2222)); 7 partList.add(new Integer(3333)); 8 partList.add("Bad Data"); // compiler error now 9 10 Iterator<Integer> elements = partList.iterator(); 11 while (elements.hasNext()) { 12 Integer partNumberObject = elements.next(); 13 int partNumber = partNumberObject.intValue(); 14 15 System.out.println("Part number: " + partNumber); 16 } 17 } 18 } The sample code that illustrates the concept of generic array list for Iteration and Boxing is as follows: for (Integer partNumberObj:partList) { int partNumber = partNumberObj; // Demos auto unboxing System.out.println("Part number: " + partNumber); }]

So here we have our for loop, so we're saying Integer partNumberObj of our partList. We can actually do the autounboxing by assigning int partNumber = partNumberObj;. And then we print out our partNumber, but that autounboxing does make it a lot easier and a lot cleaner in our code for the next person that's going to come along and take a look at it.

Implementing a TreeSet

Learning Objective

After completing this topic, you should be able to

- *describe how to implement a TreeSet using the Set interface in the Java Collection application programming interface or API*

1. Using the Set interface

When coding in Java, many times, you'll make use of lists, which are ordered collections of elements. And one point of a list is that you can have duplicate elements in your collection. The `Set` interface is similar to a list, but has three main differences. First, the `Set` can't have duplicate elements. Second, and unlike a list, a set is unordered. And third, sets don't have indexes, so you can't access the elements by using an index value. And we'll see in the demo how to use a `for` loop in order to go through the items in a set. And while a `Set` interface doesn't order your elements, Java does have a `TreeSet` class that you can use that will order your elements for you. Let's take a look at a demo application where we make use of a `TreeSet` class. So the first thing we need to do is we need to import our `Set` interface and the `TreeSet` class. So we `import java.util.Set` and `import java.util.TreeSet`. So now we can go down into the `main` method of our application. And let's create an instance of our `TreeSet` class. And we'll just call it `Set`, and it will be a set of `String` objects, and we'll name it `set = new TreeSet<>()`.

*[The setExample - NetBeans IDE 8.0.2 window is displayed and it includes menus such as File, Edit, View, and Navigate. Below these menus, a toolbar with options such as Undo, Redo, and Start is displayed. In addition, the NetBeans IDE 8.0.2 window contains two panes. The left pane is divided into two sections. The top section contains the Projects, Files, and Services tabs and the Projects tab is selected by default. The bottom section includes the SetExample - Navigator tab. The Projects tab contains the setExample parent node, which is already expanded. This node contains the Source Packages subnode, which further contains the setExample subnode. The setExample subnode is expanded and includes the SetExample.java subnode. The SetExample.java subnode is selected by default. Consequently, the corresponding information is displayed on the SetExample.java tabbed page in the top section of the right pane. The SetExample tabbed page contains the following code, which is partially displayed: 4 * and open the template*

*in the editor 5 */ 6 package setexample; 7 8 9 /** 10 * 11 * @author Admin 12 */ 13 public class SetExample { 14 15 /** 16 * @param args the command line arguments 17 */ 18 public static void main(String[] args) { 19 // TODO code application logic here 20 In the bottom section of the right pane, the Output - setExample (run) tab is displayed. The presenter enters the following code starting at line #8 on the SetExample.java tabbed page: 8 import java.util.Set; 9 import java.util.TreeSet; Next the presenter scrolls down to the main method where he enters the following code at line #21: 21 Set<String> set = new TreeSet<>();]*

So that generates our new instance. And then we can make use of the `add` method to add strings to our set. So we call `set.add`. And we'll add in the string `("one")` to `set.add` string of `("two")` and `set.add` string of `("three")`. And I'm just going to call `set.add("three");` once more. Remember, we said that sets cannot have duplicate elements. So let's just see what happens. We've made that call to add another element called...with the string `("three")`. And now we'll print those items out in our set to the screen. And we can do that using a `for` statement. We'll iterate through the items in our set and we'll use `System.out.println` to actually print out the items.

[The setExample - NetBeans IDE 8.0.2 window is displayed. On the Set.Example.java tabbed page, the presenter enters the following code starting at line #23: 23 set.add("one"); 24 set.add("two"); 25 set.add("three"); 26 set.add("three"); 27 28 for(String item:set) { 29 System.out.println("Item: " + item); 30 } 31 32 }]

Right, so if we run this...and in our Output window, we can see that we have our items – "one", "three", and "two". So one of the things about sets is that we can't have duplicate items. And even though we called `set.add` and put...and added in or tried to add in another string with `("three")` in it, the set wouldn't allow it. So we still only have three elements in our collection or in our `Set` collection. And since we're making use of a `TreeSet` class, it actually took care of putting them in order. So we have "one", "three", and "two"...that string "three" will come before the string "two". So it adheres to that other...or other property about tree sets is that they should be in order. So the main things we need to remember about sets are that they are unordered unless you make use of a `TreeSet` class, and you can't access them using an index. So in this example, we used a `for` loop. And unlike a list, you can't have duplicate elements.

[The setExample - NetBeans IDE 8.0.2 window is displayed. The Set.Example.java tabbed page includes the following code: 19 public static void main(String[] args) { 20 // TODO code application logic here 21 Set<String> set = new TreeSet<>(); 22 23 set.add("one"); 24 set.add("two"); 25 set.add("three"); 26 set.add("three"); 27 28

for(String item:set) { 29 System.out.println("Item: " + item); 30 } 31 32 } The presenter clicks the Start option on the toolbar and the following output is displayed on the Output - setExample (run) tabbed page in the bottom section of the right pane: Item: one Item: three Item: two BUILD SUCCESSFUL (total time: 0 seconds)]

Implementing a TreeMap

Learning Objective

After completing this topic, you should be able to

- *describe how to implement a TreeMap using the Map interface in the Java Collection application programming interface or API*

1. Using the Map interface

The `Map` interface is used when you want to have a collection that stores key-value pairs. And a key is a unique identifier for each element in a collection. So it doesn't have to be a number. It could be a string or a different data type when your value is simply that; it's a value that happens to be stored with your key. Now they are also known as "associative arrays". And if you've used them in other programming languages before, that's what you might have known them as. And a map is good for tracking things such as parse lists and their descriptions. So we have a quick table down here that has key-value pairs, where the Key is 101 for the first one and our Value is Blue Shirt, which would be a string. The `Map` interface doesn't extend the `Collection` interface because it represents mappings and not a collection of objects.

[Heading: Implementing a TreeMap. A table containing an example of key-value pairs is displayed. This table contains three rows and the following two columns: Key and Value. For the key 101, the Value is Blue Shirt. For the key 102, the Value is Black Shirt For the key 103, the Value is Gray Shirt. Next a flowchart representing the hierarchy of map types is displayed. This flowchart includes the following five blocks, which are displayed in three tiers: <Interface> Map, <Interface> SortedMap, Hashtable, HashMap, and TreeMap. The third tier contains the TreeMap block, which is connected to the <Interface> SortedMap in the second tier. The second tier contains the <Interface> SortedMap, Hashtable, and HashMap blocks. All these three blocks are connected to the <Interface> Map block in the first tier.]

So some of the key implementation classes of our `Map` interface include things like a `TreeMap`, which is a map where your keys are automatically sorted. We also have a `Hashtable`. And that's a classic associative array that comes with your key and values. And hash tables are synchronized, which is good if you're dealing with multiple threads in your application. We also have a `HashMap`. And that's an

implementation just like a `Hashtable` except that it accepts null keys and values. And also hash maps are not synchronized. Let's take a look at a quick demo of a `Map` or a `TreeMap` instance. So the first thing we need to do is we need to import our `Map` interface and our `TreeMap` class. So `import java.util.Map;` and we want to `import java.util.TreeMap;`.

[The MapExample - NetBeans IDE 8.0.2 window is displayed. This window includes menus such as File, Edit, View, and Navigate. Below these menus, a toolbar with options such as Undo, Redo, and Start is displayed. In addition, the NetBeans IDE 8.0.2 window contains two panes. The left pane is divided into two sections. The top section contains the Projects, Files, and Services tabs. The Projects tab is selected by default. The bottom section includes the MapExample - Navigator tab. The right pane is also divided into two sections and in the bottom section, the Output - setExample (run) tab is displayed. The Projects tab in the left pane contains the MapExample parent node, which is selected by default. As a result, the corresponding MapExample.java tabbed page is displayed in the top section of the right pane. The MapExample.java tabbed page includes the following code, which is partially visible: 1 / 2 * To change this license header, choose License Headers in Project Properties. 3 * To change this template file, choose Tools|Templates 4 * and open the template in the editor. 5 */ 6 package mapexample; The presenter enters the following import statements starting at line #7: 7 import java.util.Map; 8 import java.util.TreeMap;]*

With that added to our application, we go down to the `main` method now. So the first thing we'll do is we'll create an instance of a `TreeMap` class and we'll `Map`...and we'll say that our `Map` is going to be made up of a `String` or two strings. So our key-value pairs will both be `String`. We're going to call it `partList` that equals a `new TreeMap<>()`. So next, let's go about adding items to our `TreeMap` collection. Now when you're doing that, we don't call an `add` method, we actually have a `put` method. So we have `partList.put`. And since it is made up of two strings in our key-value pair, the first one's going to be `"s001"`, it will be our key. And our value will be `"Blue Polo Shirt"`. Let's add another one, so `partList.put`, `"s002"`. And we're going to have `"Black polo Shirt"`.

*[The MapExample.java tabbed page is displayed in the MapExample - NetBeans IDE 8.0.2 window. This tabbed page contains the following code, which is partially visible: 7 import java.util.Map; 8 import java.util.TreeMap; 9 10 /** 11 * 12 * @author Admin 13 */ 14 public class MapExample { 15 16 /** 17 *@param args the command line arguments 18 */ 19 public static void main(String[] args) { 20 //TODO code application logic here 21 22 } The presenter enters the following code starting*

at line #21: 21 Map<String, String> partList = new TreeMap<>(); 22 23 partList.put("s001", "Blue Polo Shirt"); 24 partList.put("s001", "Black polo Shirt");]

And let's add a third one, so `partList.put`, `"h001"`. And this time, it will be a `"Duke Hat"`. Now we'll add another one, so `partList.put`. But this time, I'm going to use the same key, so `"s002"`. And this will allow me to overwrite that element. Since we're using this exact same key, the new value that we're going to add here, so `"Black t-shirt"`, that will actually overwrite the `"Black polo Shirt"` that we had previously. Next what I want to do is I want to get a `Set` of our keys. So we'll create a new `Set<String> keys` instance. And we can do that by calling `partList.keySet()`.

[The MapExample.java tabbed page is displayed in the MapExample - NetBeans IDE 8.0.2 window. The presenter enters the following code starting at line #25: 25 partList.put("h001", "Duke Hat"); 26 27 partList.put("s002", "Black t-shirt"); 28 29 Set<String> keys = partList.keySet();]

So that will return a set of all our keys in our `partList TreeMap`. And I need to add that to my `import` statement, so `import java.util.Set;` there we go. And now we can print out a title, so `System.out.println`. And we'll just say `("--- Part List ---")`. And in order to iterate through our items in our `TreeMap`, we're going to use a `for` loop, so `for(String key:keys)`. And we're going to print out `System.out.println`, so we'll put in `"Part #"`. And that will be the `key` that we're using. Then we'll also put a space in there and add in a call to `partList.get`, and we'll pass in our `(key)` value, our `(key)` variable.

[The MapExample.java tabbed page is displayed in the MapExample - NetBeans IDE 8.0.2 window. The presenter scrolls up the code to the import statements section and enters the following import statement at line #9: 9 import java.util.set; Then he scrolls down to the main method and adds the following code starting at line 32: 32 System.out.println("--- Part List ---"); 33 for(String key:keys) { 34 System.out.println("Part #" + key + " " + partList.get(key));]

So that will actually be how we get the value in our tree list, it is received or returned by calling the `get` method and passing in our `(key)` value. Let's see why this `for` statement is not working, probably because I used the capital F. Okay, this should work now, we can run it. And it will generate our `TreeMap` instance and then print out all the elements in our `TreeMap`. So we'll run it. And here we have our "--- Part List ---", "Duke Hat", "Blue Polo Shirt", "Black t-shirt". And you can see that we can have the key that is not an integer, which a lot of databases, for example, have their keys...or their primary keys use numbers that increment. In this

case, we have `key`, that is a `String`. We have...starts with a character, H for hats, S for shirts, followed by three digits. So you can make use of...if you have different keys...different ways of generating your keys, a map would be a good way of organizing your information. So we've seen in this how to generate or create a `TreeMap`, we've looked at the `put` method, which is how you add elements to your `TreeMap` collection, and then we used the `get` method in order to retrieve that information from the collection.

[The MapExample.java tabbed page is displayed in the MapExample - NetBeans IDE 8.0.2 window. This tabbed page contains the following code: 1 / 2 * To change this license header, choose License Headers in Project Properties. 3 * To change this template file, choose Tools|Templates 4 * and open the template in the editor. 5 */ 6 package mapexample; 7 import java.util.Map; 8 import java.util.TreeMap; 9 import java.util.set; 10 11 /** 12 * 13 * @author Admin 14 */ 15 public class MapExample { 16 17 /** 18 *@param args the command line arguments 19 */ 20 public static void main(String[] args) { 21 //TODO code application logic here 22 Map<String, String> partList = new TreeMap<>(); 23 24 partList.put("s001", "Blue Polo Shirt"); 25 partList.put("s001", "Black polo Shirt"); 26 partList.put("h001", "Duke Hat"); 27 28 partList.put("s002", "Black t-shirt"); 29 30 Set<String> keys = partList.keySet() 31 32 System.out.println("--- Part List ---"); 33 for(String key:keys) { 34 System.out.println("Part #" + key + " " + partList.get(key)); 35 36 } 37 38 } 39 40 } 41 The presenter clicks the Start option on the toolbar. As a result, the following output is displayed in the Output – MapExample (run) tabbed page: Run: --- Part List --- Part #h001 Duke Hat Part #s001 Blue Polo Shirt Part #s002 Black t-Shirt BUILD SUCCESSFUL (total time: 0 seconds)]*

Implementing a Deque

Learning Objective

After completing this topic, you should be able to

- *describe how to implement an ArrayDeque using the Deque interface in the Java Collection application program interface or API*

1. Using the Deque interface for ArrayDeque

The `Deque` interface is actually a child interface of collection, so just like sets and lists. And a queue is often used to track asynchronous message request so that can be processed in order, while `stack` can be very useful for traversing a directory tree or similar structures. A `Deque` is a double-ended queue. So essentially, this means that a `Deque` can be used as a queue, where we have first in, first out operations, we have `add(e)`, `remove()` methods. Or we can use it as a `stack`, where we have last in and first out operation, so we have `push(e)` and `pop()` methods. In this demo, we'll look at how we can use a `Deque` as a `stack`. So the first thing we need to do is we need to input our interface in our class. So `import java.util.Deque;` and we want to `import java.util.ArrayDeque;` so now we go down to our main method, and we want to create an instance of a `Deque`. So `Deque`, and we're going to say it's made up of `<String>` elements, and we'll call it `stack`, and that equals a `new ArrayDeque<>()`.

*[The TestStrack - NetBeans IDE 8.0.2 window is displayed. The menu bar includes the following menus: File, Edit, View, Navigate, Source, Refactor, Run, and Debug. A toolbar with buttons such as New, Save, and Run is displayed below the menu bar. This window includes the left pane and right pane. The left pane includes two sections. The first section includes the following tabs: Projects, Files, and Services. The Projects tab is open and includes the TestStrack node, which further includes the following subnodes: Source Packages and Libraries. The Source Packages subnode includes the teststrack subnode and it includes the TestStrack.java class file. The second section includes the TestStrack - Navigator tab. In the right pane, the TestStrack.java tabbed page is displayed and it includes the following tabs: Source and History. The Source tab is open and it includes the following code: 4 * and open the template in the editor. 5 */ 6 package teststrack; 7 8 9 /** 10 * 11 * @author Admin 12 */ 13 public class testStrack { 14 15 /** 16 * @param args the command line arguments 17 */ 18 public static void main(String[] arg) { 19 // TODO*

code application logic here Below this tabbed page, the Output pane is displayed. The presenter enters the following code on line# 8 and line# 9 of the TestStrack.java tabbed page: 8 import java.util.Deque; 9 import java.util.ArrayDeque; He then scrolls down and enters the following code on line# 22 in the main method: 22 Deque<String> stack = new ArrayDeque<>();]

Now we can add items to our `Deque` by calling the `push` method, so `stack.push` we'll do `("one")`. We'll also do `stack.push("two");` and `stack.push("three")`. Now in order to iterate through the items in our `Deque`, I'm going to get the size of our `Deque`. We'll go `int size = stack.size()`, so we can call the `size()` method. And we're going to `-1` from that since...if we get the size of our `Deque` and we have, say, for example in this case, we have three elements, the size is three, but the number of times we actually will get the...it's zero base, so we will get zero element, first element, second element, so that's why we have to `-1`. We use that in a `while` statement, so `while (size >= 0)`.

[The TestStrack - NetBeans IDE 8.0.2 window is displayed. The presenter enters the following code from line# 23 in the main method: 23 stack.push("one"); 24 stack.push("two"); 25 stack.push("three"); 26 27 int size = stack.size() -1; 28 while (size >= 0) {]

We're going to print out our...we're going to print out the next item in our `stack`, `System.out.println`, and we'll call `(stack.pop())`. We'll also decrement our `size--;`. So now if we run our application, you see that we've printed out "three", "two", "one". So even though "three" was the last one added, when we call the `pop` method, it's the first one that come off our queue, so that's why it goes "three", "two", "one. So we've seen that how you make use of the `push` and `pop` methods of your `stack`. And we made use of an `ArrayDeque` to make it our `Deque` be, or act, or behave like a `stack` instead of a queue.

[The TestStrack - NetBeans IDE 8.0.2 window is displayed. The presenter enters the following code from line# 29: 29 System.out.println(stack.pop()); 30 size--; 31 } He then clicks the Run button on the toolbar and the Run Project warning box is displayed with the following message: One or more projects were compiled with errors. Application you are running may end unexpectedly. This warning box also includes Always run without asking checkbox and the Run Anyway and Cancel buttons. He clicks the Run Anyway button and the following result is displayed in the Output section: three two one BUILD SUCCESSFUL (total time: 3 seconds)]

Ordering Collections

Learning Objective

After completing this topic, you should be able to

- *describe how to use the Comparable and Comparator interfaces to sort Java collections*

1. Sorting Java collections

When you're looking to sort elements in a collection, you need to have a way to compare one element to another. So it could be by name, or by ID number, or even a combination of properties. The collections API provide two interfaces for ordering element, `Comparable` and `Comparator`. `Comparable` is implemented in a class and it provides a single sorting option for your class. `Comparator` on the other hand enables you to create multiple sorting options. And as you plug in, you design the option whenever you want. And both interfaces can be used with sorted collections such as tree sets and tree maps. Let's take a look at two applications and make use of the `Comparable` and the `Comparator` interfaces. First up we have the comparable interface. So we have a ComparableStudent.java class that I'll open, and it implements our `Comparable` interface. So we have `public class ComparableStudent implements Comparable<ComparableStudent>`. Now if we scroll down to the bottom, we can see that it has a `compareTo` method that will be used in order to `compareTo` all the elements together. And we'll go back up and we will notice that the interface is designed using generics. So we have `implements Comparable`, and then within their angle brackets, we're passing in the type of object that's going to be compared. Now if we go back down to our `compareTo` method,

[The NetBeans IDE 8.0.2 window is displayed. The menu bar includes the following menus: File, Edit, View, Navigate, Source, Refactor, Run, and Debug. This window includes the left pane and right pane. The left pane includes two sections. The first section includes the following tabs: Projects, Files, and Services. The Projects tab is open and includes the TestComparable and TestComparator nodes. The TestComparable node further includes the following subnodes: Source Packages and Libraries. The Source Packages node includes the testcomparable subnode and it further includes the ComparableStudent.java and TestComparable.java class files. The TestComparator further includes the following subnodes: Source

Packages and Libraries. The Source Packages node includes the testcomparator subnode and further includes the following class files: Student.java, StudentSortGpa.java, StudenSortName.java, and TestComparator.java. The StudenSortName.java file is selected by default. The second section of the left pane includes the main - Navigator tab and it includes the TestComparable node with the main(string [] args) subnode. The right pane includes two sections. The first section is empty and second section displays the Output pane. The presenter double-clicks the ComparableStudent.java class file and in the first section of the right pane, the ComparableStudent.java tabbed page is displayed. The ComparableStudent.java tabbed page includes the following tabs: Source and History. The Source tab is open and the following code is displayed: 1 / 2 * To change this license header, choose License Headers in Project Properties. 3 * To change this template file, choose Tools|Templates 4 * and open the template in the editor. 5 */ 6 package testcomparable; 7 8 /** 9 * 10 * @author Admin 11 */ 12 public class ComparableStudent implements Comparable<ComparableStudent> { 13 private String name; 14 private long id = 0; 15 private double gpa = 0.0; The presenter highlights the code on line #12. He then scrolls down and the following code is displayed: 17 public ComparableStudent(String name, long id, double gpa) { 18 this.name = name; 19 this.id = id; 20 this.gpa = gpa; 21 } 22 23 public String getName() {return this.name;} 24 public long getId(){return this.id;} 25 public double getGpa(){return this.gpa;} 26 27 public String toString() { 28 String result = "Name: " + this.name + " " + "ID: " + this.id + " " + "GPA: " + this.gpa; 29 return result; 30 } 31 public int compareTo(ComparableStudent s) { 32 int result = this.name.compareTo(s.getName()); 33 if (result > 0) {return 1;} 34 else if (result < 0) { return -1; } 35 else {return 0;} 36 } 37 38 }]*

we have a number of `if` statements. And the `if` statements are included to show how the comparisons take place, so you can just return a `result`. But in this case, when you're using the `compareTo` method, you need to have results returning you either a negative number, which means that our element comes before the current element that's being compared; if it's a positive number, then it comes after the current element. And if it returns `0`, that means that this element is equal to the current element. So we have our `if` statement; so `if (result > 0)`, and we can see that when we pass in our `(ComparableStudent s)`, it's going to compare this current instance that we're calling the method on. We're comparing it to another instance of the same class. So in this case, we're calling, `this.name.compareTo`, it's going to compare the names of our two `ComparableStudent` objects. Now in case where the collection has equivalent values, you can replace the code, you can return even different numbers. If you want to have more than just comparing it by `name`, if you want to compare it by GPA, which will be another way of another property on this object, you could add

codes so that if the result is equal to zero, you would do another set of `if` statements to do another check. So let's look at our main method for our program.

[The ComparableStudent.java file is open in the NetBeans IDE 8.0.2 window is displayed. The presenter highlights the following code from line# 31: 31 public int compareTo(ComparableStudent s) { 32 int result = this.name.compareTo(s.getName()); 33 if (result > 0) {return 1;} 34 else if (result < 0) { return —1; } 35 else {return 0;}]

So I'll open up TestComparable.java. And I also have imported `java.util.Set` and `java.util.TreeSet`. And all we've done is we've created a new `TreeSet` called the `studentList`. We've added three students, `"Thomas Jefferson"`, `"John Adams"`, and `"George Washington"`. We've given them IDs and grade point averages. Now we just simply have to call or make use of our `for` loop. In our loop, we're going to call each of the `Comparable` students in the `studentList` and print them out. Now since our tree sets are ordered...to begin with, when those items are added, our `compareTo` method gets called. And it determines where in the collection that that new element...or new item should be placed. So our `for` loop just iterates through all those comparable students, let's **Run** this. You can see that even though we've added `"Thomas Jefferson"` first, then `"John Adams"`, then `"George Washington"`, when we run it, the list has "George Washington" listed first, then "John Adams", and then "Thomas Jefferson" because they're using alphabetical order, G, J, and T. Let's look at an example using the `Comparator` interface. And this time we're going to have a Student.java class.

[The ComparableStudent.java file is open in the NetBeans IDE 8.0.2 window. The presenter double-clicks the TestComparable.java class file in the left pane. In the right pane, the TestComparable.java tabbed page is displayed and it includes the following code: 1 / 2 * To change this license header, choose License Headers in Project Properties. 3 * To change this template file, choose Tools|Templates 4 * and open the template in the editor. 5 */ 6 package testcomparable; 7 8 import java.util.Set; 9 import java.util.TreeSet; 10 11 12 /** 13 * 14 * @author Admin 15 */ 16 public class TestComparable { The presenter highlights the following codes on lines #8 and line #9: 8 import java.util.Set; 9 import java.util.TreeSet; He then scrolls down and the following code is displayed: 18 /** 19 * @param args the command line arguments 20 */ 21 public static void main(String [] args) { 22 Set<ComaparableStudent> studentList = new TreeSet<>(); 23 24 studentList.add(new Comparab1eStudent("Thomas Jefferson", 1111, 3.8)); 25 studentList.add(new Comparab1eStudent("John Adams", 2222, 3.9)); 26 studentList.add(new Comparab1eStudent("George Washington", 3333, 3.4)); 27 28 for (ComparableStudent student:studentList) { 29 System.out.println(student); 30 }*

31 } 32 33 } Next he right-clicks the TestComparable node in the left pane and the shortcut menu with the options such as New, Build, Clean and Build, and Run are displayed. He selects the Run option. As a result, in the Output pane, the following output is displayed: Name: George Washington ID: 3333 GPA: 3.4 Name: John Adams ID: 2222 GPA: 3.9 Name: Thomas Jefferson ID: 1111 GPA: 3.8 BUILD SUCCESSFUL (total time: 0 seconds) The presenter double-clicks the Student.java class file in the left pane. In the right pane, the Student.java tabbed page is displayed.]

So here's our `Student` class, it just has the `name`, `id`, and `gpa`. So it's very similar to our `ComparableStudent` class, only this time we don't have the `compareTo` method. We actually have separate StudentSortGpa.java and StudentSortName.java classes. So I'll open one up, you'll see that our `StudentSortName` class `implements Comparator`. And again, we're using generics. And the type of class that we're passing in is a `<Student>`. And we have one single method, it's just called `compare` and it's passing two `Student` objects. In this case, it's going to compare `Student s1` and `Student s2`. Again, in this one, we're going to compare them based on their name. So we just see if...`Student s1`, how it compares to `Student s2`. We return that `result`...if it's minus one or plus one, we return that result in order to determine how to sort them. And `if` the `(result !=0)`, then we `return 0;` so that takes care of all our sorting. Let's look at our TestComparator.java main method.

[The Student.java file is open in the NetBeans IDE 8.0.2 window. The presenter double-clicks the Student.java class file in the left pane. In the right pane, the Student.java tabbed page is displayed and it includes the following code: 1 / 2 * To change this license header, choose License Headers in Project Properties. 3 * To change this template file, choose Tools|Templates 4 * and open the template in the editor. 5 */ 6 package testcomparable; 7 8 /** 9 * 10 * @author Admin 11 */ 12 public class Student { 13 private String name; 14 private long id = 0; 15 private double gpa = 0.0; 16 17 public Student(String name, long id, double gpa) { 18 this.name = name; 19 this.id = id; 20 this.gpa = gpa; 21 } 22 23 public String getName() {return this.name;} 24 public long getId(){return this.id;} 25 public double getGpa(){return this.gpa;} He then opens the StudentSortName.java class file from the Projects tab in the left pane. In the right pane, the StudentSortName.java tabbed page is displayed and it includes the following code: 1 /* 2 * To change this license header, choose License Headers in Project Properties. 3 * To change this template file, choose Tools|Templates 4 * and open the template in the editor. 5 */ 6 package testcomparator; 7 8 import testcomparator.Student; 9 10 import java.util.Comparator; 11 /** 12 * 13 * @author Admin 14 */ 15 public class StudentSortName implements Comparator<Student>{ 16 public int compare(Student s1, Student s2) { 17 int result =*

s1.getName().compareTo(s2.getName()); 18 if (result !=0) {return result;} 19 else { 20 return 0; 21 } 22 } 23 } The presenter opens the TestComparator.java class file from the Projects tab in the left pane. In the right pane, the TestComparator.java tabbed page is displayed.]

So in this class, I have imported `java.util.Comparator`, `java.util.Collections`, `java.util.List`, and `java.util.ArrayList`. So here we have a new `studentList`, which is going to be a `new ArrayList<>` of `<Student>` objects. But then I also create two `Comparator` classes, we have `sortName` and `sortGpa` are the `new StudentSortName()` and `new StudentSortGpa()` classes. After that we create our list of students, so we have `studentList.add new Student`. We'll add our three new students. And then I can call `Collections.sort`. And when I call the `sort` method, I pass in the collection that I want to sort, which cases our `ArrayList` called `studentList`. And I can pass in how I want to compare them. The first time I'm going to use our `sortName`, so it would sort our collection based on our `Comparator`, where we'll sort them by name.

[The TestComparator.java class file is open in the NetBeans IDE 8.0.2 window. This page displays the following code: 1 / 2 * To change this license header, choose License Headers in Project Properties. 3 * To change this template file, choose Tools|Templates 4 * and open the template in the editor. 5 */ 6 package testcomparator; 7 8 import java.util.Comparator; 9 import java.util.Collections; 10 import java.util.List; 11 import java.util.ArrayList; 12 13 /** 14 * 15 * @author Admin 16 */ He then scrolls down and the following code is displayed: 17 public class TestComparator 18 19 /** 20 * @param args the command line arguments 21 */ 22 public static void main(String [] args) { 23 List<Student> studentList = ArrayList<> (3); 24 Comparator<Student> sortName = new StudentSortName(); 25 Comparator<Student> sortGpa = new StudentSortGpa(); 26 27 studentList.add(new ComparableStudent("Thomas Jefferson", 1111, 3.8)); 28 studentList.add(new ComparableStudent("John Adams", 2222, 3.9)); 29 studentList.add(new ComparableStudent("George Washington", 3333, 3.4)); 30 31 Collections.sort(studentList, sortName); 32 System.out.println("--- Students sorted by Name ---"); 33 for (Student student:studentList) { 34 System.out.println(student); 35 }]*

I then have another `for` loop that prints out our `studentList` by going through each `Student` in our collection. And then after that, I call `Collections.sort` again. This time we're passing in our `studentList` and a new `sort` method, `sortGpa`. And again, we just have another `for` loop that goes through each `Student` in our array and then prints them out. So let's **Run** this one. Here you can see we have "Students sorted by Name", we have "George Washington", "John

Adams", "Thomas Jefferson". And then we sort them by GPA. So "3.9", "3.8", and "3.4" GPA is set up to go in descending order. And that's how you can implement your comparisons in your classes when they are going to be used in collections. So you implement the `Comparable` interface, then you override the `compareTo` method when you're only going to provide one way to sort your data. If you have multiple sorting options, then you implement the `Comparator` interface and override the `compare` method. You can then call `Collections.sort` to sort your collection.

[The TestComparator.java class file is open in the NetBeans IDE 8.0.2 window. The presenter scrolls down and the following code is displayed: 37 Collections.sort(studentList, sortGpa); 38 System.out.println("--- Students sorted by GPA ---"); 39 for (Student student:studentList) { 40 System.out.println(student); 41 } 42 43 } He then right-clicks the TestComparator node from the Project tab in the left pane and selects the Run option from the shortcut menu that is displayed. As a result, in the Output pane, the following output is displayed: --- Students sorted by Name --- Name: George Washington ID: 3333 CPA: 3.4 Name: John Adams ID: 2222 CPA 3.9 Name: Thomas Jefferson ID: 1111 CPA: 3.8 --- Students sorted by GPA --- Name: John Adams ID: 2222 CPA: 3.9 Name: Thomas Jefferson ID: 1111 CPA: 3.8 Name: George Washington ID: 3333 CPA: 3.4 BUILD SUCCESSFUL (total time: 0 seconds)]

Using the Builder Pattern

Learning Objective

After completing this topic, you should be able to

- *describe how to use the builder pattern to create a Java class*

1. Working with the builder design pattern

In this video, we're going to take a look at the Builder design pattern. And we're going to do that by using a `Person` class and having a collection of people. So to start with our `Person` class, it's going to have attributes like `name`, `age`, and the `address`. And we're going to create it by using, as I said, the Builder pattern. And that's going to generate a collection of people for our example. Now you can use this in an app that's going to be used for contacting people via mail, phone, or e-mail. Or, you know, given a list of people, you can query for certain groups. And some of the groups that we might want to do queries for could be if we are looking for drivers that have to be people over the age of 16. If we're looking for draftees, which would mean that we have to have males that are between 18 and 25 years of age. Of course, that might be a little dated now. Of course, we can have male and female draftees. And we could also be looking for pilots, which means that we're looking for people between 23 and 65 years of age. So our groups...the queries we have for our groups, all have limitations or specifications as certain parts of our class that we're going to be looking at.

Now here we have our `Person` class. And the properties that we have are `givenName` and `surName`, which are both strings. They also have an `age`. We have a `gender` for our `Person` class and it also has `eMail`, `phone`, `address`, `city`, `state`, and `code`, the typical properties you would have of this type of class. So we looked at our `Person` class and we've seen what properties it has. And there are quite a few there. And so if we were to support constructors that allowed us to generate `Person` objects using any or all of those fields, we have quite a selection. So the use of the Builder pattern will allow us to set the properties of the `Person` object that we want to create, and a method that makes it quite a bit easier to read and to actually code. One of the ways we can do this is by including a `Builder()` method on our `Person` class. And we can then chain together methods that allow us to set the properties of our fields.

[Heading: Using the Builder Pattern. The following code sample explains the properties defined in the Person class: 9 public class Person { 10 private String givenName; 11 private String surName; 12 private int age; 13 private Gender gender; 14 private String eMail; 15 private String phone; 16 private String address; 17 private String city; 18 private String state; 19 private String code; The following code sample explains the Builder pattern: 260 people.add(261 new Person.Builder() 262 .givenName("Betty") 263 .surName("Jones") 264 .age(85) 265 .gender(Gender.FEMALE) 266 .email("betty.jones@example.com") 267 .phoneNumber("211-33-1234") 272 .build() 273);]

So on our code, we have a `new Person` class that calls `Builder()`. And we're passing in a `givenName`, a `surName`, the `age`, the `gender`, `email`, `phoneNumber`. And then we call `.build()`, which creates our `Person` object and returns it so that we can add it to our collection. So that's part of our Builder pattern. It allows you to create your objects with a lot of flexibility. Your class doesn't have to support a lot of constructors and support all the fields in the class. You can go about passing in the data piece by piece, and then request an actual instance of that particular type of class in order to add it to our collection. So this is a fairly common pattern that you can make use of when you're working with collections.

Collection Iteration and Lambdas

Learning Objective

After completing this topic, you should be able to

- *describe how to iterate through a collection using a Lambda expression*

1. Using Lambda expressions

When you have a collection of elements, you'll face moments whenever you want to access all or just a subset of items in that collection in order to do things like print out all the details or you want to make use of their properties as part of another set of programming logic. So let's take a look at how iteration works and how lambdas can help with that task. So we've the `forEach` method and that's been added now to all collections in the Java language. And this makes iteration so much easier to work with and provides a lot of benefits. And the example on the screen merely prints out all the `Person` instances in our `List`. So our collection interface extends the iterable interface and it's the iterable interface that defines our `forEach` method. So in our case, we have a `List<Person>` objects and the variable name is `pl`. So we just do a called a `pl.forEach` and in this case, we're passing in a quick lambda expression that will print out information about each `Person` in our `List`. Now within our `forEach` method, we can add our own code blocks or as I mentioned, we have our own lambda statement that is printing out our information.

[Heading: Collection Iteration and Lambdas. The following code is an example of iteration with forEach method: 9 public class RoboCallTest06 { 10 11 public static void main(String[] args){ 12 13 List<Person> pl = Person.createShortList () ; 14 15 System.out.println ("\n=== Print List ===") ; 16 pl.forEach(p -> System.out.println(p)); 17 18 } 19 }]

Now the `forEach` method on its own or when you're using it on a collection will go over each and every item in that collection. But you won't always want to use that in your code. Sometimes you only want certain elements that meet specific criteria. So that's where you can add the `stream()` method to your statement, and that opens up a whole host of new operations that you can do on your collections. And one of them is the use of the `filter` method. So in our code, we have our `filter` method and it takes a predicate as a parameter. And in our case, our

predicate is a lambda expression, and it uses that to filter out the results. So only the collection elements that match our predicate criteria will be returned to our `forEach` method. So we've `pl.stream()`, so instead of `pl.forEach`, which we had before, we've our `pl.stream()` and then we chain `.filter` on top of that passing in their predicate, which sets up our criteria. And from there, when we chain another `.forEach` method, that `forEach` method will only get those elements that meet the criteria. And then we can go ahead and print out that the information about them or make use of them in another way.

[Heading: Collection Iteration and Lambdas. The following code sample explains the use of Stream and Filter methods: 10 public class RoboCallTest07 { 11 12 public static void main(String[] args){ 13 14 List<Person> pl = Person.createShortList () ; 15 RoboCall05 robo = new RoboCall05 (); 16 17 System.out.println ("\n=== Calling all Drivers Lambda ==="); 18 pl.stream() 19 .filter(p —> p.getAge() >= 23 && p.getAge() <= 65) 20 .forEach(p —> robo.roboCall(p)) ; 21 22 } 23 }]

So this is really a big improvement on looping, making use of the `stream()` method. And in this case, our collection statement with the `stream()` really describes what's happening. We're taking the collection, we're going to filter out certain elements, and return the results. So the code itself is much cleaner than it would've been years ago using Java to go through all the elements in your collection. And here's another example and this code once again shows that a lambda expression could be stored in a variable and used or reused later. So we have our `pl.stream().filter` and then we're passing in `(allPilots)`. And on the previous slide, we had our lambda expression located here. But we can assign that lambda expression...a predicate is an actual object that we can create using our lambda expressions. So earlier in our code on line 18, we've predicate where we're using a `Person` object type, so `allPilots =` and then we have our lambda expression. So we can take that, use it later on with our `filter` statement. And that still will filter out all the elements in our collection that meets our certain criteria, and that gets passed into our `forEach` method.

[Heading: Collection Iteration and Lambdas. The following code sample is an another example, which explains the Stream and Filter methods: 10 public class RoboCallTest08 { 11 12 public static void main(String[] args) { 13 14 List<Person> pl = Person.createShortList () ; 15 RoboCall05 robo = new RoboCall05 (); 16 17 // Predicates 18 Predicate<Person> allPilots = 19 p —> p.getAge() >= 23 && p.getAge() <= 65; 20 21 System.out.println("\n=== Calling all Drivers Variable ==="); 22 pl.stream().filter(allPilots) 23 .forEach(p —> robo.roboCall(p)); 24 }]

The Stream API

Learning Objective

After completing this topic, you should be able to

- *describe the Stream Application Program Interface or API and how it can be used to process Java collection elements in parallel*

1. Using the Java Stream API

This video will have us discussing the Java Stream API. And the Stream API is found in the `java.util.stream` namespace. And you use a stream in order to permit method chaining, which is really calling multiple methods in a single statement. The first characteristic of streams that you should know about is that they're immutable; their state doesn't change after they've been constructed. Next, it's important to realize that once elements have been consumed, they are no longer available from the stream. As well, the chain of operations is applied only once within a stream. And lastly, streams can be run in serial, which is the default setting, but they can also be executed in parallel. Now your stream pipeline has three main parts. There is a source such as a collection, or a file, or another stream. Then there is also an intermediate operation, which you may or may not have. So a filter will be an example of an intermediate operation. And the last part of our stream pipeline is the terminal operation. And our `forEach` method is an example of the terminal. And that takes the result of our intermediate operations and applies the final code block to our stream. So let's look at one of the intermediate operations say, and for us, it's going to be the `filter` method.

So our `stream()` will take our source, and it's most likely a collection object, and it converts that to our pipeline. Our `filter` method can then be applied to our pipeline. And it checks to see which items in the pipeline meet the criteria in our filter. And our filter accepts our `Predicate` lambda that sets out all our specific criteria. So you can see in our sample code that we have a `tList.stream()`. And then we chain a call to our `filter` method that has our `Predicate` lambda, that's going to look to see if our object `getState()` method returns a string that `equals("CA")`. Now only the elements that pass this filter will then be used in our `forEach` statement.

[Heading: The Stream API. The Stream class consists of Immutable data, and can only be used once, and then tossed. The Filter method uses Predicate lambdas to

select items. The following code sample explains the tList.stream class and the filter method: 15 System.out.println("\n== CA Transaction Lambda =="); 16 tList.stream() 17 .filter(t -> t.getState(),equals("CA")) 18 .forEach (SalesTxn::printSummary);]

Method References and Method Chaining

Learning Objective

After completing this topic, you should be able to

- *describe how to call an existing method using a method reference and how methods can be chained in Java Lambda expressions*

1. Using a method-referencing code

Now in many situations, a method reference can be substituted for a lambda expression. So instead of our lambda expression calling `t.printSummary()`, we could, in fact, use a method-referencing code, `(SalesTxn::printSummary)` instead. So when should you forego using lambdas and go with method references. Well, first, you could do it when you're referencing a static method. Next, if you reference an instance method where your syntax would be the `ObjectName::instanceMethodName`, or if you reference an instance method of a particular type and you have the syntax of `ClassName::instanceMethodName`. And lastly, you could use a method reference when you're referencing a constructor. So those are the four types...that would give you reason to switch from lambdas to method references. In your pipeline, you can have one or more intermediate operations along with a terminal operation. And each of these intermediate and terminal statements are called from our single statement since they're chained together. And as you can see from our example, we can even have multiple calls to the same method such as `filter`.

[Heading: Method References and Method Chaining. The following sample code explains how the lambda expression calls a class method: .forEach(t -> t.printSummary()) Alternatively, use the following method reference: .forEach(SalesTxn::printSummary)); The following method reference can be used to call a static method: ContainingClass::staticMethodName The method reference can also be used to reference an instance method of an arbitrary object of a particular type such as String::compareToIgnoreCase. In addition, the method reference can be used to reference a constructor such as ClassName::new. The following sample code explains how multiple calls can be made to the same method: 21 tList.stream() 22 .filter (t -> t.getState().equals("CA")) 23 .filter(t ->

t.getBuyer().getName() 24 .equals("Acme Electronics")) 25 .forEach(SalesTxn::printSummary);]

If you want to think of analogies, you can think about this as these statements are really similar to SQL statements where we have our `WHERE` clauses. The syntax is, of course, different, but the idea is very similar. We just have `WHERE` clause, after `WHERE` clause, after `WHERE` clause, that we can keep adding to our SQL statements the same way as we have multiple intermediate operations in our pipeline. And when you're doing your chaining, your `Predicate` can be as complex or as simple as you like. So our first example on our slide has a compound statement in our single `filter` method call. While the second example is exactly the same code; it's merely breaking up the filter and possibly making it more readable. So the style is really upto you or your company's coding standards.

[Heading: Method References and Method Chaining. The following sample code explains the use of compound logical statements: 15 System.out.println("\n== CA Transactions for ACME =="); 16 tList.stream() 17 .filter(t -> t.getState().equals("CA") && 18 t.getBuyer ().getName ().equals("Acme Electronics")) 19 .forEach(SalesTxn: :printSummary); 20 21 tList.stream() 22 .filter(t -> t.getState() .equals("CA")) 23 .filter(t -> t.getBuyer() .getName() 24 .equals ("Acme Electronics")) 25 .forEach(SalesTxn: :printSummary);]

Exercise: Java SE8 Programming fundamentals

Learning Objective

After completing this topic, you should be able to

- *learn the features of Java interfaces, the inner anonymous class, and Java streams*

1. Exercise overview

We've seen quite a bit information. Let's take a few minutes to practice using a HashMap object and a Deque object in an application.

In this exercise, you'll practice identifying the key features and characteristics of Java interfaces, the anonymous inner class, and Java stream. This involves the following tasks:

• Identifying the features of various Java interfaces
• Identifying the characteristics of the anonymous inner class
• Recognizing features of the Java stream

2. Java SE8 Programming features

Question

Which of the following best describes a characteristic of Java interfaces?

Options:

1. Java interfaces are similar to abstract classes
2. Classes cannot be interface referenced
3. Java interfaces contain only dynamic fields
4. Java interfaces list the methods with implementation code

Answer

Option 1: *Correct. Java interfaces are similar to abstract classes where they outline the public methods that are coded by any class that implements the interface.*

Option 2: *Incorrect. Any class that implements a Java interface can be referenced using that interface. This is useful as objects used by the different classes can be processed in the same way.*

Option 3: *Incorrect. Java interfaces can also contain constant fields.*

Option 4: *Incorrect. Java interfaces list the methods without the implementation of code. This means that the curly braces and their contents are omitted.*

Correct answer(s):

1. Java interfaces are similar to abstract classes

Question

Which of the following is a characteristics of the anonymous inner class?

Options:

1. Defines a class within the code
2. Allows other classes to use it
3. Decreases the level of encapsulation
4. Separates logical code

Answer

Option 1: *Correct. An anonymous inner class is a class that is defined within the code instead of being defined in a separate file. This enables logical grouping of code, increases encapsulation, and makes the code more readable.*

Option 2: *Incorrect. The anonymous inner class can only be used by the specific class it resides in.*

Option 3: *Incorrect. Using the anonymous inner class increases code encapsulation as it allows the inner class to only be used by the class that it is in.*

Option 4: *Incorrect. The anonymous inner class allows logical code to be grouped in one place. This also improves the readability of the code.*

Correct answer(s):

1. Defines a class within the code

Question

Which of the following classes is used to apply the implementation code for the Map interface?

Options:

1. `HashTable`
2. `HashSet`
3. `LinkedList`
4. `ArrayDeque`

Answer

Option 1: *Correct. The* `HashTable` *class is used to implement the Map interface. In addition, the* `HashMap` *and* `TreeMap` *interfaces are used for this purpose.*

Option 2: *Incorrect. The* `HashSet`, `TreeSet`, *and* `LinkedHashSet` *classes are used to implement the Set interface.*

Option 3: *Incorrect. The* `ArrayList` *and* `LinkedList` *classes are used to implement the List interface.*

Option 4: *Incorrect. The* `ArrayDeque` *is used to implement the Deque interface.*

Correct answer(s):

1. `HashTable`

Question

Match the appropriate features with the type of interface.

Options:

A. Provides first in, first out or FIFO and last in, first out or LIFO operations
B. Utilizes the push and pop methods
C. Ideal for key-value pairs
D. Unordered list of unique elements
E. Ordered collection of elements

Targets:

1. Deque
2. Map
3. Set
4. List

Answer

The Deque interface provides for both LIFO and FIFO operations and utilizes the push and pop methods. The Map interface is used for key-value pairs. The Set interface represents unordered lists with unique elements. The List interface represents an ordered collection of elements.

The Deque interface can take the form of a stack or a queue providing for both LIFO or FIFO operations. The Deque interface represents either a

queue or a stack. A queue provides FIFO operations utilizing add and remove methods. A stack provides LIFO operations utilizing the push and pop methods.

The Map interface represents a collection that stores multiple key-value pairs. The key is a unique identifier for each element in the collection.

A Set interface represents an unordered list of unique elements and has no index. Elements are accessed through iterating through them, as opposed to a List interface where an index allows access to any point.

A List interface represents an ordered collection of elements, which allow duplicate elements. Elements can be added, removed, or overwritten anywhere within the index.

Correct answer(s):

Target 1 = Option A, Option B

Target 2 = Option C

Target 3 = Option D

Target 4 = Option E

Question

Which of the following is a feature of the stream application program interface or API?

Options:

1. Enables method chaining
2. Allows stream muting
3. Enables the stream elements to be available always
4. Runs only in serial order

Answer

Option 1: *Correct. The stream API permits method chaining where multiple methods can be called within a single statement.*

Option 2: *Incorrect. Streams are immutable, this means that their state does not change after they have been constructed.*

Option 3: *Incorrect. When elements are consumed, they are no longer available from the stream.*

Option 4: *Incorrect. Streams can run in serial or parallel order and chain operations can only occur only one time on a particular stream.*

Correct answer(s):

1. Enables method chaining